Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches

Viet Ba Dang[1], Farnoud Farahmand[1], Michal Andrzejczak[2],
Kamyar Mohajerani[1], Duc Tri Nguyen[1] and Kris Gaj[1]

[1] Cryptographic Engineering Research Group,
George Mason University
Fairfax, VA, U.S.A.
[2] Military University of Technology
Warsaw, Poland

Abstract. Performance in hardware has typically played a major role in differentiating among leading candidates in cryptographic standardization efforts. Winners of two past NIST cryptographic contests (Rijndael in case of AES and Keccak in case of SHA-3) were ranked consistently among the two fastest candidates when implemented using FPGAs and ASICs. Hardware implementations of cryptographic operations may quite easily outperform software implementations for at least a subset of major performance metrics, such as speed, power consumption, and energy usage, as well as in terms of security against physical attacks, including side-channel analysis. Using hardware also permits much higher flexibility in trading one subset of these properties for another. A large number of candidates at the early stages of the standardization process makes the accurate and fair comparison very challenging. Nevertheless, in all major past cryptographic standardization efforts, future winners were identified quite early in the evaluation process and held their lead until the standard was selected. Additionally, identifying some candidates as either inherently slow or costly in hardware helped to eliminate a subset of candidates, saving countless hours of cryptanalysis. Finally, early implementations provided a baseline for future design space explorations, paving a way to more comprehensive and fairer benchmarking at the later stages of a given cryptographic competition. In this paper, we first summarize, compare, and analyze results reported by other groups until mid-May 2020, i.e., until the end of Round 2 of the NIST PQC process. We then outline our own methodology for implementing and benchmarking PQC candidates using both hardware and software/hardware co-design approaches. We apply our hardware approach to 6 lattice-based CCA-secure Key Encapsulation Mechanisms (KEMs), representing 4 NIST PQC submissions. We then apply a software-hardware co-design approach to 12 lattice-based CCA-secure KEMs, representing 8 Round 2 submissions. We hope that, combined with results reported by other groups, our study will provide NIST with helpful information regarding the relative performance of a significant subset of Round 2 PQC candidates, assuming that at least their major operations, and possibly the entire algorithms, are off-loaded to hardware.

Keywords: Post-Quantum Cryptography · hardware · software/hardware co-design · FPGA · System on Chip · ASIC · Key Encapsulation Mechanism · digital signature · public-key · ARM · NEON

1 Introduction

Hardware benchmarking has played a major role in all recent cryptographic standardization efforts, such as the AES, eSTREAM, SHA-3 [11, 32, 44, 45], and CAESAR contests [17, 18]. With the emergence of commonly-accepted hardware application programming interfaces (APIs) [38], development packages [34, 37], specialized optimization tools [31, 23], new design methodologies based on High-Level Synthesis (HLS) [35, 36], and mandatory hardware implementations in the final round of the CAESAR contest [17], the percentage of initial submissions implemented in hardware grew from 27.5% in the SHA-3 contest [30] to 49.1% in the CAESAR competition [18, 29]. In Round 2, all AES, all SHA-3, and all but one CAESAR candidates had at least one hardware implementation reported by the end of the evaluation process.

In almost all cases, candidates performing particularly well in hardware were identified quite early during the evaluation process. For example, Keccak led in terms of speed in hardware already in Round 2 of the SHA-3 contest. It outperformed 13 remaining Round 2 candidates and the old standard SHA-2. AEGIS-128 was identified as one of the three fastest authenticated ciphers in Round 2 of the CAESAR contest when implemented using high-performance FPGAs, Virtex-6, Virtex-7, Stratix IV, and Stratix V. It outperformed at least 25 other candidates and the current standard AES-GCM. At the same time, during each contest, several candidates were identified as particularly costly, slow, or cumbersome to implement in hardware. Examples included Mars during the AES contest; BMW, ECHO, and SIMD during the SHA-3 competition; HS1-SIV, POET, and OMD in the CAESAR contest. The early identification of hardware inefficiency helped to focus the effort of the cryptographic community on more promising candidates, potentially saving countless hours of cryptanalysis.

Hardware vs. software. Cryptographic algorithms are routinely implemented using both software and hardware. By software, we mean implementations that can be executed using processors. These processors may vary from low-cost low-power embedded processors, such as ARM Cortex-M4, to high-performance general-purpose microprocessors, such as Intel Core i7, with Haswell microarchitecture, supporting Advanced Vector Extensions 2 (AVX2) and the AES New Instructions (AES-NI). The common feature is that all of these processors are typically programmed using high-level programming languages, such as C. Code written in these languages is portable among different processor types. Software implementations can be further optimized by using assembly language programming, involving instructions specific to a given processor (or more accurately to its Instruction Set Architecture (ISA)). Assembly language programs are not easily portable among processors based on different ISAs.

By hardware, we mean implementations that can be executed using Field Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), Programmable Logic (PL) of System on Chip FPGAs (SoC FPGAs), Application-Specific Standard Products (ASSPs), etc. The common feature is that most of these implementations are developed using hardware description languages (HDLs), such as VHDL and Verilog. These languages differ substantially from high-level programming languages by introducing the concepts of an entity, connectivity, concurrency, and timing. HDL source code is transformed by a synthesis tool to a netlist composed of basic logic components and connections among these components. Because of its generic nature, HDL code can be easily ported among different technologies, such as FPGAs and ASICs. ASIC implementations are faster, use less power, and require less physical area. FPGA implementations have the advantage of less expensive development tools, much shorter design cycle, and reconfigurability, understood as an ability to change the function of all internal building blocks and connections among them, even after a given integrated circuit has been deployed in actual products.

Low cost and short development cycle are decisive factors making FPGAs more suitable for benchmarking and ranking candidates during the evaluation period. Reconfigurability supports the algorithm and parameter agility, making FPGAs more frequently used than ASICs during the early stages of deployment of PQC in real products. The relative performance of cryptographic algorithms in FPGAs has been shown correlated to their relative performance in ASICs [32]. At the same time, this correlation is not guaranteed to hold across multiple classes of cryptographic transformations, e.g., it is not guaranteed to work equally well for hash functions and PQC algorithms. Therefore, both FPGA and ASIC benchmarking studies are essential.

Although software implementations are likely to be dominant during the first phase of deploying PQC standards in real applications, hardware implementations will inevitably follow. They are likely to start from hardware accelerators for constrained environments, such as smart cards and Internet of Things devices. Low-cost low-power processors used in such applications may not be able to keep up with the increased demands for computational power and energy usage. Thus, these processors may need to be extended with hardware accelerators. In the medium term, high-performance security processors enhanced with new PQC standards will emerge. These processors will be optimized to process in hardware all the algorithms associated with secure communication (such as those used in the post-quantum versions of TLS, IPSec, IKE, and WTLS/WAP protocols) and secure storage. Finally, in the longer-term, support for new instructions, enabling the efficient and side-channel resistant implementations of PQC standards, is likely to be added to the most popular processor ISAs. Co-processors for such instructions are, effectively, hardware implementations of PQC. Taking into account that the new PQC standards are likely to remain in use for decades, all of the mentioned above use cases should be given considerable weight. In particular, the performance of a given algorithm in hardware may affect its long-term performance in software, on processors equipped with new specialized instructions. Even if Round 2 hardware implementations are not a final word in terms of the algorithm performance, they provide the first glimpse into each candidate's suitability for hardware acceleration. They also establish an open source-code base on which more optimized implementation and implementations protected against side-channel and fault attacks can be built in Round 3 and beyond.

High-speed vs. lightweight. Assuming the use of the same technology, hardware implementations outperform software implementations using at least one, and typically multiple metrics, such as speed, power consumption, energy usage, and security against physical attacks. They also allow much higher flexibility in trading one subset of these metrics for another. From the point of view of benchmarking and ranking of candidates, such flexibility may become a curse, especially taking into account that no two metrics are likely to have a simple linear dependence on each other. A practical solution to this problem is to focus during the evaluation process on two major types of implementations: high-speed and lightweight.

In high-speed implementations, the primary target is speed. For PQC schemes, this target amounts to minimizing the execution times of major operations involving the public and private key, respectively. For Key Encapsulation Mechanisms (KEMs), these operations are encapsulation and decapsulation; for digital signature schemes, signature verification and generation; for public-key encryption (PKE), encryption and decryption. The time of key generation may also play a major role in the case when a public-private key pair cannot be reused for security reasons. The resource utilization is secondary. Still, hardware designers typically aim at achieving the Pareto optimality, in which any further improvement in speed comes at the disproportionate cost in terms of resource utilization. The primary advantage of high-speed implementations is that they reveal the inherent potential of a given algorithm for parallelization. As long as the resource-utilization limit is sufficiently high, this limit does not affect the ranking of algorithms. As a result, the ranking is strongly correlated with the features of algorithms themselves and is not substantially influenced by any additional assumptions and technology choices. Additionally, only high-speed hardware implementations may effectively compete with optimized software implementations targeting high-performance processors with vector instructions (e.g., AVX2).

In lightweight implementations, the primary targets are typically minimum resource utilization and minimum power consumption, under the assumption that the execution time does not exceed a predefined maximum. Another way of formulating the goal is to achieve minimum execution time, assuming a given maximum budget in terms of resource utilization, power consumption, or energy usage. The maximum budget on resource utilization is related to the cost of implementation; the budget on power assures correct operation without overheating or devoting additional resources to cooling. The maximum energy usage affects how long a battery-operated device can function before the next battery recharge. In the context of the standardization process for cryptographic algorithms, the mentioned above maximum budgets are very hard to select. Any change in these thresholds may favor a different subset of candidates. With new standards remaining in use for decades, timing, cost, and power requirements of new and emerging applications are very challenging to predict.

Additionally, changes in technology significantly affect which hardware architectures meet particular constraints. For example, an architecture capable of accomplishing the execution time of 0.1 seconds (or below), under a certain power or energy budget, may substantially change with the improvements in technology. As a result, the majority of current limits are selected somewhat arbitrarily by different designers, or left undefined in their reports. Consequently, the ranking of PQC candidates based on their lightweight implementations, especially those developed by different groups, is extremely challenging and assumption-dependent. These rankings have little to do with the parallelization allowed by each algorithm, as most of the operations must be executed sequentially due to the small resource budget. The primary feature of algorithms these implementations reveal is the number and complexity of its distinct elementary operations. Each major operation infers an additional functional unit, increasing resource utilization and power consumption. Additionally, lightweight hardware implementations can outperform only software implementations targeting specific low-cost low-power embedded processors, such as Cortex-M4.

In the case of FPGA implementations, resource utilization is a vector, such as (#LUTs, #flip-flops, #DSP units, #BRAMs). No single element of this vector can be expressed in terms of other elements. As a result, imposing a resource limit implies specifying the values of all components of this resource vector. One possible approach may be to choose the resources of the smallest FPGA of a given low-cost FPGA family. However, FPGA families and their resources change over time, so this limit has only a physical meaning during the limited time, covering the evaluation period, and may lose its significance just a few years after the standard is published and deployed. Finally, the same FPGA device may also need to accommodate any overhead associated with countermeasures against side-channel attacks. At the same time, this overhead or even effective countermeasures may remain unknown at the time of the candidates' evaluation.

As a result, in this paper, we focus on the development, benchmarking, and ranking of high-speed implementations. At the same time, we do our best to summarize lessons learned from the development of lightweight implementations by other groups.

Speeding-up the development process. Traditionally software and hardware benchmarking were conducted separately by different groups of experts, equipped with different knowledge and tools. Even the units for expressing speed were different – cycles per byte for software and megabits per second for hardware. For PQC algorithms, this approach is hard to maintain. These algorithms are simply too complex and too different from the current state-of-the-art in public-key cryptography to permit the development of optimized pure-hardware implementations for a significant fraction of Round 2 candidates by any single group within the time frame imposed by the NIST evaluation process.

Two approaches to overcome the long development time have emerged. The first is software/hardware co-design [22, 76]; the second is the use of HLS [14, 22, 57].

Software/hardware co-design has been used for years in the industry and studied extensively in academia, with the goal of reaching performance targets using a shorter development cycle than that typical for hardware-only implementations. To the best of our knowledge, no benchmarking of software/hardware co-designs was reported during any previous cryptographic competitions. As a result, multiple problems specific to cryptographic contests, such as the choice of the most representative platform(s) and the fairness of software/hardware partitioning schemes, have never been addressed. It should be clearly stated that software/hardware benchmarking is not intended as a replacement for pure-hardware benchmarking. On the contrary, applying this approach to selected Round 2 candidates and developing a library of hardware accelerators for major operations of these candidates will make it much easier to develop hardware-only implementations in subsequent rounds. Although the software/hardware co-design approach can be used to realize both high-speed and lightweight implementations, in this paper, we focus on its application to high-speed designs.

Within the proposed framework, one of the first issues to address is the choice of the appropriate platform. In particular, we need a computing platform allowing fast communication across the software/hardware boundary. We also need the suitable prototyping board, as the timing measurements had to be performed experimentally, and the computing platform had to be well-suited for attempting various software/hardware partitioning schemes. The choice of a suitable device and prototyping board is addressed in Section 4. With the preferred platform identified, our second major concern is the fairness of software/hardware benchmarking, especially in terms of deciding which operations within each evaluated scheme should be offloaded to hardware. In this paper, we propose a comprehensive approach to address this issue, aimed at achieving the best possible trade-off between the final performance and the required development time. This approach is described in detail in Section 4.

The second approach to substantially accelerating the development time is the use of High-Level Synthesis. This approach amounts to refactoring a software implementation in C or C++ in such a way that this implementation can be used as an input to a High-Level Synthesis tool, such as Vivado HLS or LegUp, capable of automatically transforming such an implementation to HDL code. The result is a pure hardware implementation obtained based on the code written in a traditional programming language (typically C and C++). This language is turned into a high-level hardware description language using synthesis directives encoded using pragmas and specific coding techniques aimed at exposing the potential for parallelization and resource utilization reduction.

This approach has been demonstrated to substantially reduce the development time of Round 2 and Round 3 CAESAR candidates. At the same time, it provided an almost identical ranking of candidates in terms of throughput and throughput to area ratio. However, taking into account significant differences between the complexity and underlying operations of secret-key authenticated ciphers and public-key PQC schemes, the use of HLS for benchmarking of PQC candidates remains controversial. The common perception is that obtained results are significantly worse in terms of both speed and resource utilization compared to manual HDL coding. However, our preliminary research indicates that, with a proper approach, the penalty in terms of the execution time in clock cycles can be made negligibly small. Only the penalty in terms of resource utilization and clock frequency remains. The former overhead affects only the secondary metric in high-speed designs; the latter can be kept in a similar range for multiple candidates. As a result, the use of high-level synthesis when applied to high-speed designs should remain an active area of research, and should not be dismissed upfront before more case studies are performed.

Choice of FPGA family. One of the major concerns is the NIST recommendation to focus on hardware benchmarking using the Xilinx Artix-7 FPGA family. This recommendation appeared in several NIST presentations related to Round 2 of the NIST standardization process, e.g., during PQCrypto 2019 in May 2019 and the Second PQC Standardization Conference in August 2019. We believe that, in its current form, this recommendation is counterproductive, and it impedes rather than supports fair and comprehensive hardware and software/hardware benchmarking.

Let us start by explaining what an FPGA family is and what influence does it have on an evaluation process. FPGA family is a set of FPGA devices sharing the same internal structure and the same process technology (also known as technology node or process node), described by a number related to the size and density of transistors that can be fabricated using a given manufacturing process. With the steady improvements in process technology, described by Moore's Law, the maximum capacity and speed of FPGA devices have been steadily increasing while their prices have remained approximately the same. Every new generation of FPGA devices of a particular vendor receives a unique name, referred to as a family name. Every family consists of multiple devices with various distinct sizes to match the needs of different applications. All devices of a particular family share the same internal architecture and process technology but differ in terms of the number of resources of a particular type, such as Look-Up Tables (LUTs), flip-flops (FFs), block memories, and digital signal processing units (DSP units) or multipliers. Most vendors release both low-cost families (such as Xilinx Artix-7) and high-performance families (such as Xilinx Virtex-7). Most of them also release mid-range families, such as Xilinx Kintex-7. The maximum amount of resources available in the largest device of a low-cost family is naturally significantly smaller than the equivalent amount in the largest device of a high-performance family (e.g., over 5 times smaller for Artix-7 vs. Virtex-7).

Additionally, in recent years, FPGA vendors started releasing new types of programmable devices that enhance Programmable Logic of traditional FPGAs with the Processing System based on a hardwired embedded processor, such as ARM. Since this processor is custom designed, it takes full advantage of a given technological process and operates at a clock frequency significantly higher than Programmable Logic. With a fast processor and an efficient interface between this processor and Programmable Logic, these devices are ideal for software/hardware co-designs targeting high-speed. Although these types of devices appear under multiple commercial names, they are often collectively referred to as System on Chip FPGAs (SoC FPGAs). The first family of this type was Xilinx Zynq-7000, released in 2011, based on ARM Cortex-A9 embedded processors.

Hardware designs are described in hardware description languages. HDL code is typically identical for all FPGA families. As opposed to software, where each processor may require different optimized assembly language code, no such concepts exist for hardware. As a result, it is straightforward to synthesize the same HDL code targeting various FPGA families from various vendors, as long as the maximum capacity of the largest device of a given family is not exceeded.

Giving preference to the Xilinx Artix-7 family has several undesired consequences summarized below:

1. Artix-7 is a low-cost FPGA family. As such, it is not very suitable for high-speed implementations. Hardware resources of even the largest device of this family are often insufficient to demonstrate the full potential for parallelizing operations a given PQC algorithm. Thus, the use of Artix-7 makes perfect sense for benchmarking lightweight implementations but may lead to suboptimal results for high-speed implementations.

2. Artix-7 is a traditional FPGA, and not an SoC FPGA. As a result, the only way to develop a single-chip software/hardware implementation using Artix-7 is the use

of so-called "soft" processor cores, i.e., processors implemented using programmable logic. Soft processors compatible with Artix-7 include MicroBlaze and lightweight versions of RISC-V. All of them operate at much lower clock frequency than hardwired embedded processors of SoC FPGAs.

3. Artix-7 is unsuitable for HLS designs. Such designs typically take significantly more resources than designs based on writing code manually in HDL. As a result, assuming the Pareto optimization for high-speed, they are unlikely to fit in the largest Artix-7 FPGA.

4. Artix-7 is a relatively old FPGA family, released by Xilinx in 2010. By the time of the release of the PQC standard, this family will be at least 12 years old. While still relatively popular for low-cost applications, this family does not represent the state-of-the-art in FPGA technology.

5. It is not customary to base ranking of candidates in cryptographic contests on results obtained for a single family of a single vendor. Although Xilinx is the largest developer of FPGAs and SoC FPGAs, Intel comes a strong second, and other vendors, such as Microchip and Lattice Semiconductor, also develop FPGAs suitable for implementing cryptographic algorithms. During the SHA-3 competition, the results were reported for seven FPGA families from two major vendors, Xilinx and Altera. During the CAESAR contest, four Xilinx families and four Altera families were employed. For all of these families, results were generated based on the same HDL code. There was no need to purchase multiple tools or boards. Free or trial versions of tools were sufficient. The designs ended with the generation of post-place-and-route reports, which correctly described the worst-case performance of any particular instance of the given FPGA device.

6. Based on the authors' experiences, multiple reviewers of papers devoted to implementations of Round 2 PQC candidates treated the NIST's choice of Artix-7 as an absolute requirement. Submissions not complying with this requirement were subject to rejection or requests for major revisions. As a result, a noble goal of making the results more comparable with one another was turned into a reason for suppressing or delaying the publication of relevant results.

Taking these concerns into account, our recommendation for Round 3 is to encourage reporting results for at least the following FPGA families:

1. For lightweight hardware implementations and lightweight software/hardware implementations based on soft processor cores: Xilinx Artix-7 (for compatibility with Round 2 results) and Intel Cyclone 10 LP.

2. For lightweight software/hardware implementations based on the use of hard processor cores: Xilinx Zynq 7000-series and Intel Cyclone V SoC FPGAs.

3. For high-speed hardware and high-speed software/hardware implementations: Zynq Xilinx UltraScale+ and Intel Stratix 10 SoC.

One of the reasons for selecting Zynq Xilinx UltraScale+, even for pure hardware implementations that do not require SoC capabilities, is the support for these devices by the free version of the Xilinx toolset, called Vivado HL WebPACK, which is sufficient to generate all required benchmarking results. Xilinx Virtex-7 UltraScale+ FPGAs, which could be considered as a natural candidate, are not supported by the same free version of tools. The Zynq Xilinx UltraScale+ family is also recommended for high-speed software/hardware implementations based on the use of hard processor cores because of moderate cost of suitable prototyping boards and the availability of a free Benchmarking

Setup for Software/Hardware Implementations of PQC Schemes, developed at George Mason University [21].

2 Previous Work

Table 1: Reported Hardware Implementations

Algorithms	High-Speed	Lightweight
Lattice-based : Encryption/Key Exchange		
CRYSTALS-KYBER	[79], [14]^H, CERG	[12], [13]*, [1], [27]
FrodoKEM	[39], [14]^H, [19]	[12], [13]*
LAC	[79], CERG	–
NewHope	[14]^H, [28], [80], [79], [41], CERG	[12], [13]*, [1], [27]
NTRU	[14]^H, CERG	–
NTRU Prime	CERG	–
Round5	[19], [4], CERG	[3]
SABER	[14]^H, [19], [54], [62]	[27]
Three Bears	–	–
Isogeny-based : Encryption/Key Exchange		
SIKE	[49], [53], [20]	[53]
Code-based : Encryption/Key Exchange		
BIKE	[5], [60]	–
Classic McEliece	[74], [14]^H	–
HQC	–	–
LEDAcrypt	[14]^H	[40]
NTS-KEM	–	–
ROLLO	–	–
RQC	–	–
Lattice-based : Digital Signature		
CRYSTALS-DILITHIUM	[14]^H	[12], [13]*
FALCON	–	–
qTESLA	[14]^H	[12], [13]*, [75]
Symmetric-based : Digital Signature		
Picnic	[42]	–
SPHINCS+	[14]^H	–
Multivariate : Digital Signature		
GeMSS	–	–
LUOV	–	–
MQDSS	[14]^H	–
Rainbow	[25]	–

^{H} design developed using the High-Level Synthesis (HLS) approach
* extended version of [12]

Hardware and software/hardware implementations of Round 2 PQC candidates reported to date are summarized in Table 1. The PQC candidates are grouped by family and a type of scheme. All Encryption and Key Exchange schemes are listed first, followed by Digital Signature schemes. The Encryption and Key Exchange schemes have candidates from three major families: lattice-based, isogeny-based, and code-based. The Digital Signature schemes have candidates representing lattice-based, symmetric-based, and multivariate families. All implementations are classified as either High-Speed or Lightweight. However, the dividing line is not always very clear, and, in multiple cases, the authors have not used these terms explicitly by themselves.

HLS-based implementations are distinguished with the superscript ^{H}. These implementations were reported in only one paper [14]. They have been shown to give substantially different results than implementations developed using traditional Register-Transfer Level (RTL) methodology, in which HDL code is developed manually. Therefore, in this section, we focus on implementations in which a hardware part of the design was developed using traditional RTL methodology. The HLS designs are discussed separately in Section 5.4.

Eight out of 26 candidates (31%) do not have any high-speed implementation to date, and 13 (50%) do not have any RTL high-speed implementation. 17 out of 26 (65%) do not have any lightweight implementation. The coverage of the code-based family is the weakest, with only 3 out of 7 candidates (BIKE, Classic McEliece, and LEDAcrypt) implemented targeting high-speed (including only two using RTL-based approach), and only 1 out of 6 (LEDAcrypt) realized using a lightweight approach. Similarly, the multivariate family remains mostly unexplored. Only two out of four candidates have their implementations reported, including only one following the RTL methodology. The symmetric-based digital signatures have no lightweight implementations, and even among high-speed implementations, only one is the RTL-based implementation.

The coverage of the lattice-based and isogeny-based encryption/key exchange schemes is the most complete. Eight out of nine lattice-based KEMs have high-speed implementations reported. The only exception is Three Bears. Five out of these eight have, on top of that, at least one lightweight implementation. In terms of the number of various implementations, NewHope leads the way with 10 related publications, followed by CRYSTALS-KYBER, with 7, and FrodoKEM and Saber with 5. The only isogeny-based scheme, SIKE, has been thoroughly explored in hardware as well, especially taking into account the earlier implementations of the underlying key agreement scheme SIDH [47], [9], [48], [46], [50].

The coverage of lattice-based signatures is not as good as lattice-based KEMs. In particular, FALCON appears to be very difficult to implement, using either the high-speed or lightweight approach. Additionally, even in the case of CRYSTALS-DILITHIUM, somewhat surprisingly, its only high-speed implementation to date is an HLS-based design.

In Tables 2–8, we summarize major results for hardware and software/hardware implementations of KEMs. Most of the schemes are KEMs with indistinguishability under the chosen-ciphertext attack (IND-CCA). Some are PKEs with indistinguishability under the chosen-plaintext attack (IND-CPA). If an IND-CPA-secure PKE is reported, this fact is marked with a superscript ^{cpa}. All mentioned above tables have the same fields. The first two columns contain a reference to the publication and the name of the algorithm variant, respectively. The superscript ^{Z} next to the publication reference indicates the implementation using Zynq-7000 SoC FPGA. The implementations targeting Artix-7 and Zynq-7000 are grouped together because the programmable logic of both families is realized using the same technological process and composed of the same basic building blocks. In the third column, the type of implementation is indicated, with HW standing for hardware, and SW/HW standing for software/hardware. Among the software/hardware implementations, we specify the embedded processors used with the following notation: ^{RV} represents a RISC-V processor with the RV32IM ISA, i.e., RISC-V with the base 32-bit integer ISA and the standard Integer Multiplication and Division extension. ^{c} represents a custom processor, and ^{A9} a hard processor of the Zynq-7000 SoC FPGA family, namely ARM Cortex-A9. Unlike the first two options, this processor operates with the frequency significantly higher than the maximum clock frequency of programmable logic. At the same time, the transfer of control and data between the processor and the hardware accelerator contributes a non-negligible transfer overhead to all reported execution times.

Table 2: Level 1 KEMs and PKEs on Artix-7 (default) and Zynq-7000 (indicated with the superscript ^{Z})

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BR AM	Key Generation		Encaps./Enc.^{cpa}		Decaps./(Dec.+Enc.)^{cpa}	
										cycles	μs	cycles	μs	cycles	μs
									Security Level 1						
[80]^Z	NewHope-512^{cpa}	HW	HS	200	6,780	4,026	–	2	7.0	4,200	21.0	6,600	33.0	9,100	45.5
[74]	mceliece348864^{cpa}	HW	HS	106	81,339	132,190	–	0	236.0	202,787	1,920.3	2,720	25.8	12,743	120.7
[74]	mceliece348864^{cpa}	HW	HS	108	25,327	49,383	–	0	168.0	1,599,882	14,800.0	2,720	25.2	18,358	169.8
[27]^Z	Kyber-512	SW/HW^{RV}	LW	–	23,925	10,844	–	21	32.0	150,106	–	193,076	–	204,843	–
[39]	FrodoKEM-640 16x	HW	HS	172	2,587	2,994	855	16	0	204,766	1,190.5	207,269	1,212.1	209,867	1,408.5
				171	5,796	4,694	1,692	16	0						
				149	6,881	5,081	1,947	16	12.5						
[13]	Kyber-512	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	74,519	2,980.8	131,698	5,267.9	142,309	5,692.4
[27]^Z	NewHope-512	SW/HW^{RV}	LW	–	23,925	10,844	–	21	32.0	123,860	–	207,299	–	226,742	–
[13]	NewHope-512	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	97,969	3,918.8	236,812	9,472.5	258,872	10,354.9
[27]^Z	LightSaber	SW/HW^{RV}	LW	–	23,925	10,844	–	21	32.0	366, 837	–	526, 496	–	657,583	–
[1]	Kyber-512	SW/HW^{RV}	LW	59	1,842	1,634	–	5	34.0	710,000	11,993.2	971,000	16,402.0	870,000	14,695.9
[1]	NewHope-512	SW/HW^{RV}	LW	59	1,842	1,634	–	5	34.0	904,000	15,270.3	1,424,000	24,054.1	1,302,000	21,993.2
[53]	SIKEp434	SW/HW^c	HS	162	22,595	11,558	7,491	162	37.0	1,474,200	9100	2,494,800	15,400.0	2,656,800	16,400.0
[53]	SIKEp503	SW/HW^c	HS	162	22,595	11,558	7,491	162	37.0	1,733,400	10,700.0	2,932,200	18,100.0	3,126,600	19,300.0
[39]	FrodoKEM-640 1x	HW	LW	191	971	433	290	1	0	3,237,288	16,949.2	3,275,862	17,241.4	3,306,122	20,408.2
				190	4,246	2,131	1,180	1	0						
				162	4,446	2,152	1,254	1	12.5						
[53]	SIKEp434	SW/HW^c	LW	143	10,976	7,115	3,512	57	21.0	2,187,902	15,300.0	3,718,004	26,000.0	3,946,804	27,600.0
[53]	SIKEp503	SW/HW^c	LW	143	10,976	7,115	3,512	57	21.0	2,602,603	18,200.0	4,390,104	30,700.0	4,676,105	32,700.0
[13]	FrodoKEM-640	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	11,453,942	458,157.7	11,609,668	464,386.7	12,035,513	481,420.5
[5]	BIKE Level 1	HW	HS	135	1,865	589	590	0	4.0	7,370,429	54,540.0	–	–	–	–

^{Z} Design implemented on Zynq-7000

^{cpa} Design of a PKE variant resistant against Chosen-Plaintext Attack (CPA)

^{RV} co-design using RISC-V RV32IM

^{A9} co-design using ARM Cortex-A9

* Preliminary result

Table 3: Level 3 & 5 KEMs and PKEs on Artix-7 (default) and Zynq-7000 (indicated with the superscript ^{Z})

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BR AM	Key Generation		Encaps./Enc.^{cpa}		Decaps./(Dec.+Enc.)^{cpa}	
										cycles	μs	cycles	μs	cycles	μs
Security Level 3															
[74]	mceliece460896^{cpa}	HW	HS	107	38,669	74,858	–	0	303.0	5,002,044	46,704.4	3,360	31.4	31,005	289.5
[39]	FrodoKEM-976 16x	HW	HS	169	2,869	3,000	908	16	0	476,056	2,816.9	479,993	2,857.1	483,073	3,076.9
				168	6,188	4,678	1782	16	0						
				157	7,213	5,087	2042	16	19.0						
[54]^{Z}	Saber	SW/HW^{A9}	HS	125	7,400	7,331	–	28	2.0	–	3,273.0	–	4,147.0	–	3,844.0
[13]	Kyber-768	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	111,525	4,461.0	177,540	7,101.6	190,579	7,623.2
[53]	SIKEp610	SW/HW^{c}	HS	162	22,595	11,558	7,491	162	37.0	2,916,000	18,000.0	5,443,200	33,600.0	5,508,000	34,000.0
[39]	FrodoKEM-976 1x	HW	LW	189	1,243	441	362	1	0	7,560,000	40,000.0	7,480,000	40,000.0	7,714,286	47,619.0
				187	4,650	2,118	1,272	1	0						
				162	4,888	2,153	1,390	1	19.0						
[53]	SIKEp610	SW/HW^{c}	LW	143	10,976	7,115	3,512	57	21.0	4,347,204	30,400.0	8,108,108	56,700.0	8,208,208	57,400.0
[13]	FrodoKEM-976	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	26,005,326	1,040,213.0	29,749,417	1,189,976.7	30,421,175	1,216,847.0
[5]	BIKE Level 3	HW	HS	135	1,884	557	593	0	5	30,447,947	231,400.0	–	–	–	–
Security Level 5															
[80]^{Z}	NewHope-1024^{cpa}	HW	HS	200	6,781	4,127	–	2	8.0	8,000	40.0	12,500	62.5	17,300	86.5
[41]^{Z}	NewHope-1024^{cpa}	HW	HS	190	13,244	8,272	–	24	18.0	–	–	34,000	178.0	30,600^{KD}	160.0^{KD}
[13]	Kyber-1024	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	148,547	5,941.9	223,469	8,938.8	240,977	9,639.1
[13]	NewHope-1024	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	97,969	3,918.8	236,812	9,472.5	258,872	10,354.9
[27]^{Z}	Kyber-1024	SW/HW	LW	–	23,925	10,844	–	21	32.0	349,673	–	405,477	–	424,682	–
[27]^{Z}	NewHope-1024	SW/HW	LW	–	23,925	10,844	–	21	32.0	235,420	–	392,734	–	450,541	–
[28]	NewHope-1024^{cpa}	SW/HW	HS	25	26,606	26,303	–	32	1.0	357,052	14,282.1	589,285	23,571.4	756,932	30,277.3
[27]^{Z}	FireSaber	SW/HW	LW	–	23,925	10,844	–	21	32.0	1,300,272	–	1,622,818	–	1,898,051	–
[1]	Kyber-1024	SW/HW^{RV}	LW	59	1,842	1,634		5	34.0	2,203,000	37,212.8	2,619,000	44,239.9	2,429,000	41,030.4
[53]	SIKEp751	SW/HW^{c}	HS	162	22,595	11,558	7,491	162	37.0	3,742,200	23,100.0	6,188,400	38,200.0	6,658,200	41,100.0
[1]	NewHope-1024	SW/HW^{RV}	LW	59	1,842	1,634		5	34.0	1,776,000	30,000.0	2,742,000	46,317.6	2,528,000	42,702.7
[53]	SIKEp751	SW/HW^{c}	LW	143	10,976	7,115	3,512	57	21.0	7,965,108	55,700.0	13,156,013	92,000.0	14,185,614	99,200.0
[13]	FrodoKEM-1344	SW/HW^{RV}	LW	25*	14,975	2,539	4,173	11	14.0	67,994,170	2,719,766.8	71,501,358	2,860,054.3	72,526,695	2,901,067.8

^{Z} Design implemented on Zynq-7000

^{cpa} Design of a PKE variant resistant against Chosen-Plaintext Attack (CPA)

^{RV} co-design using RISC-V RV32IM

^{c} co-design using a custom processor

^{A9} co-design using ARM Cortex-A9

* Preliminary result

^{KD} total execution time of Key Generation and Decryption

The next column, Max. Freq., corresponds to the maximum clock frequency in MHz. The next five columns are used to report FPGA resource utilization, described as a vector (LUT, FF, Slice, DSP, BRAM), where the subsequent fields represent the number of look-up tables, flip-flops, slices, DSP units, and 36 kbit Block RAMs. For the last of these values, BRAM, 0.5 represents the use of an 18-kbit block RAM.

In the case of KEMs, the remaining 6 columns are used to show the execution time of Key Generation, Encapsulation, and Decapsulation, expressed in clock cycles and μs, respectively. In the cases when only results for the IND-CPA PKE are reported, the last two columns represent the sum of the execution times of Encryption and Decryption. This convention is used because the most popular transformations between an IND-CPA-secure PKE and the corresponding IND-CCA-secure KEM involve both the Decryption and Encryption operations on the receiver's side. Additionally, these two operations dominate the total Decapsulation time. For all execution times, the value in μs can be obtained by dividing the corresponding number of clock cycles by the maximum clock frequency in MHz.

In Tables 2 and 3, we summarize implementations targeting Xilinx Artix-7 FPGAs and related Xilinx Zynq-7000 SoC FPGAs. For the security level 1, six candidates - Classic McEliece, CRYSTALS-Kyber, FrodoKEM, NewHope, SIKE, and Saber - have implementations of all three operations reported. The preliminary implementation of BIKE focuses on key generation only. For security level 3, NewHope does not have a variant. For security level 5, the results are missing for Classic McEliece.

For most KEMs, the time of decapsulation is longer than the time of encapsulation. Table entries are ordered according to the time of decapsulation in μs (and, if needed, according to the decapsulation time in clock cycles).

The ranking of candidates listed in Tables 2 and 3 is very challenging to determine based on available results. First, it may be unfair to compare pure hardware implementations with software/hardware implementations. Secondly, it is hard to compare lightweight implementations with high-speed implementations, as they are optimized with different primary metrics in mind. Third, software/hardware implementations based on different processors are very challenging to compare with one another. Finally, even for implementations using exactly the same type of implementation (software/hardware) and the same type of processor (RISC-V), such as those reported in [27], the comparison may be unintentionally biased. In the specific case of [27], significantly different hardware support was provided for algorithms that can take advantage of the Number Theoretic Transform - Kyber and NewHope - vs. the algorithm that cannot - Saber. An additional, relatively minor factor is that several results for Classic McEliece and NewHope concern their IND-CPA-secure PKEs rather than IND-CCA-secure KEMs.

Taking all these factors into account, almost the only ranking that is quite clear from Tables 2 and 3 is the ranking of candidates that have results available for pure hardware implementations targeting high-speed. In this specific category, the ranking for the security level 1 is: *1.* NewHope, *2.* Classic McEliece, *3.* FrodoKEM. If we assume that a software/hardware implementation of SIKE with a custom processor is almost as efficient as a pure hardware implementation, then we can also add SIKE at position *4.* At level 3, NewHope does not have a variant, and at level 5, Classic McEliece and FrodoKEM, do not have high-speed pure hardware implementations reported.

In Tables 4 and 5, we summarize implementations targeting Xilinx Virtex-7 FPGAs. Unfortunately, the only conclusion that can be drawn from these tables is an advantage of Classic McEliece over SIKE in terms of all performance metrics other than the number of LUTs and flip-flops.

Table 4: Level 1 KEMs on Virtex-7 (default) and Virtex-6 (indicated with the superscript ^{V6})

Design	**Algorithm**	**Type**	**Target**	**Max. Freq.**	**LUT**	**FF**	**Slice**	**DSP**	**BR AM**	**Key Generation**		**Encap./Enc.^{cpa}**		**Decaps./Dec.^{cpa}**	
										cycles	μs	**cycles**	μs	**cycles**	μs
								Security Level 1							
[49]	SIKEp503	HW	HS	171	25,094	26,971	9,514	264	34.0	640,000	3,738.3	1,120,000	6,542.1	1,210,000	7,067.8
[53]	SIKEp434	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	981,180	6,900.0	1,677,960	11,800.0	1,777,500	12,500.0
[53]	SIKEp503	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	1,166,040	8,200.0	1,976,580	13,900.0	2,104,560	14,800.0
[40]^{V6}	LEDAkem-128^{o,cpa}	HW	LW	235	104	53	33	0	1.0	–	–	712,000	3,029.8	2,620,000	18,714.3
				140	2,222	658	870	0	13.0						
[53]	SIKEp434	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	2,191,781	14,400.0	3,713,851	24,400.0	3,957,382	26,000.0
[53]	SIKEp503	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	2,602,740	17,100.0	4,383,562	28,800.0	4,672,755	30,700.0

[cpa] Design of a KEM variant resistant against Chosen-Plaintext Attack (CPA)

[V6] Design implemented on Virtex-6

[o] Design for an old parameter set

Table 5: Level 3 & 5 KEMs and PKEs on Virtex-7

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BR AM	Key Generation		Encaps./Enc.^{cpa}		Decaps./(Dec.+Enc.)^{cpa}	
										cycles	μs	cycles	μs	cycles	μs
							Security Level 3								
[74]	mceliece460896^{cpa}	HW	HS	131	109,484	168,939	–	0	446.0	515,806	3,943.5	3,360	25.7	17,931	137.1
[53]	SIKEp610	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	1,962,360	13,800.0	3,654,540	25,700.0	3,711,420	26,100.0
[53]	SIKEp610	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	4,353,120	28,600.0	8,097,412	53,200.0	8,219,178	54,000.0
							Security Level 5								
[74]	mceliece6960119^{cpa}	HW	HS	130	116,928	188,324	–	0	607.0	974,306	7,500.4	5,413	41.7	25,135	193.5
[74]	mceliece6688128^{cpa}	HW	HS	137	122,624	186,194	–	0	589.0	1,046,139	7,658.4	5,024	36.8	29,754	217.8
[74]	mceliece8192128^{cpa}	HW	HS	130	123,361	190,707	–	0	589.0	1,286,179	9,901.3	6,528	50.3	32,765	252.2
[74]	mceliece6960119^{cpa}	HW	HS	141	44,154	88,963	–	0	563.0	11,179,636	79,570.4	5,413	38.5	46,141	328.4
[74]	mceliece6688128^{cpa}	HW	HS	136	44,345	83,637	–	0	446.0	12,389,742	91,034.1	5,024	36.9	52,333	384.5
[74]	mceliece8192128^{cpa}	HW	HS	134	45,150	88,154	–	0	525.0	15,185,314	113,154.4	6,528	48.6	55,330	412.3
[49]	SIKEp751	HW	HS	167	45,893	50,390	17,530	512	43.5	1,240,000	7,407.4	2,170,000	12,963.0	2,330,000	13,918.8
[53]	SIKEp751	SW/HW	HS	142	21,210	13,657	7,408	162	38.0	2,516,940	17,700.0	4,166,460	29,300.0	4,479,300	31,500.0
[53]	SIKEp751	SW/HW	LW	152	10,937	7,132	3,415	57	21.0	7,960,426	52,300.0	13,150,685	86,400.0	14,185,693	93,200.0

^{cpa} Design of a PKE variant resistant against Chosen-Plaintext Attack (CPA)

In Tables 6 and 7, we summarize the results for ASICs. ASIC performance studies have been reported in [79], [12], and [27]. All three studies were conducted using different ASIC processes and standard-cell libraries. Therefore, the obtained results cannot be compared across any two, not to mention three, publications from this list. In all three cases, the approach was the development of a lattice-based co-processor supporting at least three different IND-CCA secure KEMs. The design presented in [79] is a domain-specific vector co-processor, leveraging the extensible RISC-V architecture. This co-processor has been integrated with an open-source RISC-V microprocessor, supporting the RV32IMC ISA. The RISC-V core has been modified to recognize the custom instructions and forward them to the vector co-processor. Similarly, in [12], the domain-specific Sapphire crypto-processor is coupled with an efficient RISC-V microprocessor, supporting the RV32IM ISA. In [27], the authors proposed an enhanced RISC-V architecture that embeds a set of powerful tightly coupled accelerators to speed up lattice-based PQC. These accelerators are deeply integrated into the RISC-V pipeline. RISC-V was also extended with 28 new instructions for performing packed modular arithmetic, butterfly operation, update of Twiddle factors, update/multiplication with scaling factors, bit-reversal, hash computations, and binomial sampling.

In all three co-processors, all supported lattice-based KEMs share the same resources. Therefore, only the comparison in terms of the execution time, power consumption, and energy was possible. All implementations adopted the SW/HW co-design approach. In [79], the target was high-speed. In [12] and [27] the minimum power and energy. In Table 6, we report the execution times, and in Table 7 both energy and power consumption at a specific operating frequency, common for all schemes compared within each study.

Performance estimates for ThreeBears have been provided in [33]. The implementations are assumed to be realized in the ASIC technology, with the standard-cell library TSMC 40 LP. The target clock frequency is 100 MHz. Two accelerators, 32-bit and 64-bit, are considered. All estimates are made by using the arithmetic core from a Rambus public-key crypto accelerator and data reported in [12]. No actual implementation is attempted.

In the study reported in [12], for the TSMC 40nm library, the ranking of candidates in terms of the decapsulation time is *1.* Kyber, *2.* NewHope, *3.* Frodo-KEM. The difference between positions *1* and *2* is by a factor of 1.8 for the security level 1, and 7% for the security level 5. FrodoKEM lags behind NewHope by a factor larger than 46 for the security level 1, and 280 for the security level 5. In terms of the power usage, the differences among all three candidates are very small, and their ranking identical to that for the decapsulation time. In terms of energy usage, Kyber and NewHope are very close to each other, and FrodoKEM lags behind by more than two orders of magnitude.

In terms of comparison with ThreeBears, the estimated performance of BabyBear is better than that reported for Kyber-512 and NewHope-512, for both 64-bit and 32-bit accelerators. Similarly, the performance of MamaBear is estimated to be better than that of Kyber-768 for both types of accelerators. For PapaBear, only the 64-bit accelerator is estimated to be faster than Kyber-1024 and NewHope-1024. The 32-bit accelerator is slightly slower. However, one should keep in mind that the processor from [12] has been fully implemented, while the results in [33] are based on estimates only.

One of the advantages of ThreeBears is that its hardware implementations can share resources with implementations of classical public-key schemes, such as Elliptic Curve Cryptography and RSA. This feature may be particularly important during the transition period, when hybrid schemes, based on both classical public-key cryptography (mostly ECC) and PQC, are likely to be used concurrently. However, the importance of this resource sharing will likely diminish over time, when classical schemes are gradually phased out. On top of that, no PQC digital signature scheme based on similar big-integer arithmetic is currently under consideration for a new NIST PQC standard.

Table 6: All KEMs on ASIC

Design	Algorithm	Type	Target	Max. Freq.	Area (kGE)	SRAM (kB)	Key Gen.		Encapsulation		Decapsulation		Technology
							cycles	us	cycles	us	cycles	us	
Security Level 1 & 2													
[79]	Kyber-512	SW/HW	HS	300	979	12.00	18,556	61.9	45,886	153.0	79,989	266.6	TSMC 28 nm
[79]	NewHope-512	SW/HW	HS	300	979	12.00	18,563	61.9	44,513	148.4	84,501	281.7	
[79]	LAC-128-v3a	SW/HW	HS	300	979	12.00	107,511	358.4	189,550	631.8	281,953	939.8	
[33]^E	BabyBear	SW/HW	HS	100	145	–	24,300	243.0	36,900	369.0	60,800	608.0	TSMC 40 nm
[33]^E	BabyBear	SW/HW	LW	100	120	–	57,500	575.0	93,200	932.0	135,800	1,358.0	
[12]	NewHope-512	SW/HW	LW	72	106	40.25	52,063	723.1	136,077	1,890.0	142,295	1,976.3	
[12]	Kyber-512	SW/HW	LW	72	106	40.25	74,519	1,035.0	131,698	1,829.1	142,309	1,976.5	
[12]	FrodoKEM-640	SW/HW	LW	72	106	40.25	11,453,942	159,082.5	11,609,668	161,245.4	12,035,513	167,159.9	
[27]	Kyber-512	SW/HW	LW	45	170	465*	150,106	3,316.5	193,076	4,265.9	204,843	4,525.9	UMC 65 nm
[27]	NewHope-512	SW/HW	LW	45	170	465*	123,860	2,736.6	207,299	4,580.2	226,742	5,009.8	
[27]	LightSaber	SW/HW	LW	45	170	465*	366,837	8,105.1	526,496	11,632.7	657,583	14,529.0	
Security Level 3 & 4													
[33]^E	MamaBear	SW/HW	HS	100	145	–	46,000	460.0	65,200	652.0	98,300	983.0	TSMC 40 nm
[33]^E	MamaBear	SW/HW	LW	100	120	–	121,000	1,210.0	177,800	1,778.0	238,900	2,389.0	
[12]	Kyber-768	SW/HW	LW	72	106	40.25	111,525	1,549.0	177,540	2,465.8	190,579	2,646.9	
[12]	FrodoKEM-976	SW/HW	LW	72	106	40.25	26,005,326	361,185.1	29,749,417	413,186.3	30,421,175	422,516.3	
Security Level 5													
[79]	Kyber-1024	SW/HW	HS	300	979	12.00	39,689	132.3	81,569	271.9	136,475	454.9	TSMC 28 nm
[79]	NewHope-1024	SW/HW	HS	300	979	12.00	36,584	121.9	85,871	286.2	161,623	538.7	
[33]^E	PapaBear	SW/HW	HS	100	145	–	74,200	742.0	100,900	1,009.0	143,000	1,430.0	TSMC 40 nm
[12]	Kyber-1024	SW/HW	LW	72	106	40.25	148,547	2,063.2	223,469	3,103.7	240,977	3,346.9	
[12]	NewHope-1024	SW/HW	LW	72	106	40.25	97,969	1,360.7	236,812	3,289.1	258,872	3,595.4	
[33]^E	PapaBear	SW/HW	LW	100	120	–	206,400	2,064.0	288,400	2,884.0	368,000	3,680.0	
[12]	FrodoKEM-1344	SW/HW	LW	72	106	40.25	67,994,170	944,363.5	71,501,358	993,074.4	72,526,695	1,007,315.2	
[27]	Kyber-1024	SW/HW	LW	45	170	465*	349,673	7,725.9	405,477	8,958.8	424,682	9,383.2	UMC 65 nm
[27]	NewHope-1024	SW/HW	LW	45	170	465*	235,420	5,201.5	392,734	8,677.3	450,541	9,954.5	
[27]	FireSaber	SW/HW	LW	45	170	465*	1,300,272	28,728.9	1,622,818	35,855.5	1,898,051	41,936.6	

^{E} The results reported in [33] were estimated using the arithmetic core from a Rambus public-key crypto accelerator and data reported in [12]. No actual implementation was attempted.

All SW/HW co-designs use RISC-V.

* Numbers reported in kGE

Table 7: Power and Energy Comparison for all KEMs on ASIC

Design	Algorithm	Type	Target	Freq.	Area (kGE)	Memory (kB)	Key Generation		Encapsulation		Decapsulation		Technology
							Power (mW)	Energy (μJ)	Power (mW)	Energy (μJ)	Power (mW)	Energy (μJ)	
Security Level 1													
[79]	Kyber-512	SW/HW	HS	300	979	12.00	29.26	1.81	23.67	3.62	24.94	6.65	TSMC 28 nm
[79]	NewHope-512	SW/HW	HS	300	979	12.00	31.84	1.97	27.77	4.12	23.82	6.71	
[79]	LAC-128-v3a	SW/HW	HS	300	979	12.00	25.90	9.28	24.33	15.37	23.74	22.31	
[12]	Kyber-512	SW/HW	LW	72	106	40.25	5.77	5.97	5.12	9.37	5.69	11.25	TSMC 40 nm
[12]	NewHope-512	SW/HW	LW	72	106	40.25	5.30	4.37	5.30	10.02	5.80	11.46	
[12]	FrodoKEM-640	SW/HW	LW	72	106	40.25	6.65	1,057.65	7.01	1,129.95	6.88	1,150.83	
[27]	NewHope-512	SW/HW	LW	10	170	465*	–	–	–	–	2.42	135.03	UMC 65 nm
[27]	Kyber-512	SW/HW	LW	10	170	465*	–	–	–	–	2.58	141.41	
[27]	LightSaber	SW/HW	LW	10	170	465*	–	–	–	–	2.78	431.18	
Security Level 3													
[12]	Kyber-768	SW/HW	LW	72	106	40.25	5.28	8.19	5.19	12.80	5.86	15.52	TSMC 40 nm
[12]	FrodoKEM-976	SW/HW	LW	72	106	40.25	6.70	2,420.97	7.05	2,912.95	6.94	2,932.13	
Security Level 5													
[79]	Kyber-1024	SW/HW	HS	300	979	12.00	35.45	4.69	29.20	7.94	25.57	11.63	TSMC 28 nm
[79]	NewHope-1024	SW/HW	HS	300	979	12.00	29.36	3.58	24.53	7.02	23.57	12.70	
[12]	Kyber-1024	SW/HW	LW	72	106	40.25	5.95	12.27	5.25	16.30	5.91	19.76	TSMC 40 nm
[12]	NewHope-1024	SW/HW	LW	72	106	40.25	6.13	8.35	5.05	16.59	5.89	21.17	
[12]	FrodoKEM-1344	SW/HW	LW	72	106	40.25	6.75	6,374.45	7.10	7,050.83	7.00	7,051.21	
[27]	NewHope-1024	SW/HW	LW	10	170	465*	–	–	–	–	2.41	259.98	UMC 65 nm
[27]	Kyber-1024	SW/HW	LW	10	170	465*	–	–	–	–	2.60	307.68	
[27]	FireSaber	SW/HW	LW	10	170	465*	–	–	–	–	2.77	1335.48	

All SW/HW co-designs using RISC-V RV32IM

* Numbers reported in kGE

Table 8: All KEMs and PKEs on Zynq Ultrascale+

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BRAM	Key Gen.		Encapsulation		Decapsulation	
										cycles	us	cycles	us	cycles	us
Security Level 1															
[19]	R5ND_1KEM_0d	SW/HW	HS	260	55,442	82,341	10,627	0	2	–	–	–	19.0	–	24.0
[19]	LightSaber	SW/HW	HS	322	12,343	11,288	1,989	256	3.5	–	–	–	53.0	–	56.0
[19]	FrodoKEM-640	SW/HW	HS	402	7,213	6,647	1,186	32	13.5	–	–	–	1,223.0	–	1,319.0
Security Level 3															
[62]	Saber	HW	HS	250	45,895	18,705	–	0	2	4,320	17.3	5,231	20.9	6,461	25.8
[62]	Saber	HW	HS	250	25,079	10,750	–	0	2	5,435	21.8	6,618	26.5	8,034	32.1
[19]	R5ND_3KEM_0d	SW/HW	HS	249	73,881	109,211	14,307	0	2	–	–	–	24.0	–	33.0
[19]	Saber	SW/HW	HS	322	12,566	11,619	1,993	256	3.5	–	–	–	60.0	–	65.0
[19]	FrodoKEM-976	SW/HW	HS	402	7087	6693	1190	32	17	–	–	–	1,642.0	–	1,866.0
Security Level 5															
[19]	R5ND_5KEM_0d	SW/HW	HS	212	91,166	151,019	18,733	0	2	–	–	–	32.0	–	42.0
[41]	NewHope-1024^{cpa}	HW	HS	406	13,961	8,149	–	25	18	–	–	34,000	83.0	30,600^{KD}	75.0^{KD}
[19]	FireSaber	SW/HW	HS	322	12,555	11,881	2,341	256	3.5	–	–	–	74.0	–	80.0
[19]	FrodoKEM-1344	SW/HW	HS	417	7,015	6,610	1,215	32	17.5	–	–	–	2,186.0	–	3,120.0

All SW/HW co-designs using ARM Cortex-A53

^{cpa} Design of a PKE variant resistant against Chosen-Plaintext Attack (CPA)

^{KD} total execution time of Key Generation and Decryption

Table 9: Digital Signature Schemes on Artix-7, Kintex-7 and Virtex-7

Design	Algorithm	Type	Target	Max. Freq.	LUT	FF	Slice	DSP	BR AM	Key Gen.		Signature Verification		Signature Generation		Family
										cycles	us	cycles	us	cycles	us	
Security Level 1 & 2																
[42]	Picnic-L1-FS	HW	HS	91	90,535	23,516	25,160	0	52.5	–	–	29,600	325.6	31,300	344.3	Artix-7
[13]	qTESLA-I ^{o2}	SW/HW	LW	25*	14,975	2,539	4,173	11	14.0	4,846,949	193,878.0	38,922	1,556.9	168,273	6,730.9	
[13]	Dilithium-I	SW/HW	LW	25*	14,975	2,539	4,173	11	14.0	95,202	3,808.1	142,576	5,703.0	376,392	15,055.7	
[13]	Dilithium-II	SW/HW	LW	25*	14,975	2,539	4,173	11	14.0	130,022	5,200.9	184,933	7,397.3	514,246	20,569.8	
[75]	qTESLA-p-I	SW/HW	LW	121	7,212	4,378	2,438	15	139.0	925,431	7,648.2	946,520	7,822.5	4,165,160	34,422.8	
[25]	Rainbow-Ic ^{o1}	HW	HS	90	52,895	32,476	15,112	0	67.0	–	–	–	–	979	10.9	Kintex-7
[25]	Rainbow-Ia	HW	HS	111	27,712	27,679	8,939	0	59.0	–	–	–	–	1,980	17.8	
[42]	Picnic-L1-FS	HW	HS	125	90,037	23,105	–	0	52.5	–	–	29,600	237.0	31,300	250.0	
[25]	Rainbow-Ic ^{o1}	HW	HS	167	52,721	32,475	15,976	0	67.0	–	–	–	–	979	5.9	Virtex-7
[25]	Rainbow-Ia	HW	HS	181	27,556	27,675	7,065	0	59.0	–	–	–	–	1,980	10.9	
Security Level 3																
[13]	qTesla-III-speed ^{o2}	SW/HW	LW	25*	14,975	2,539	4,173	11	14.0	11,898,241	475,929.6	67,712	2,708.5	317,083	12,683.3	Artix-7
[13]	qTesla-III-size ^{o2}	SW/HW	LW	25*	14,975	2,539	4,173	11	14.0	11,479,190	459,167.6	69,154	2,766.2	348,429	13,937.2	
[13]	Dilithium-III	SW/HW	LW	25*	14,975	2,539	4,173	11	14.0	167,433	6,697.3	229,481	9,179.2	634,763	25,390.5	
[75]	qTESLA-p-III	SW/HW	LW	121	7,475	4,518	2,473	15	147.0	2,305,220	19,051.4	2,315,950	19,140.1	7,745,088	64,009.0	
Security Level 4 & 5																
[42]	Picnic-L5-FS	HW	HS	125	167,530	33,164	–	0	98.5	–	–	146,600	1,173.0	154,500	1,236.0	Kintex-7
[13]	Dilithium-IV	SW/HW	LW	25*	14,975	2,539	4,173	11	14.0	223,272	8,930.9	276,221	11,048.8	815,636	32,625.4	Artix-7

^{o1} Design for a parameter set withdrawn at the beginning of Round 2

^{o2} Design for a heuristic parameter set withdrawn by the submitters on Aug. 20, 2019

All SW/HW co-designs using RISC-V RV32IM

* Preliminary result

Table 10: Digital Signature Schemes on ASIC

Design	Algorithm	Type	Target	Max. Freq.	Area (kGE)	SRAM (kB)	Key Gen.		Signature Verification		Signature Generation		Technology
							cycles	us	cycles	us	cycles	us	
Security Level 1 & 2													
[12]	qTESLA-I ^{o2}	SW/HW	LW	72	106	40.25	4,846,949	67,318.7	38,922	540.6	168,273	2,337.1	TSMC 40 nm
[12]	Dilithium-I	SW/HW	LW	72	106	40.25	95,202	1,322.3	142,576	1,980.2	376,392	5,227.7	
[12]	Dilithium-II	SW/HW	LW	72	106	40.25	130,022	1,805.9	184,933	2,568.5	514,246	7,142.3	
Security Level 3													
[12]	qTesla-III-speed ^{o2}	SW/HW	LW	72	106	40.25	11,898,241	165,253.3	67,712	940.4	317,083	4,403.9	TSMC 40 nm
[12]	qTesla-III-size ^{o2}	SW/HW	LW	72	106	40.25	11,479,190	159,433.2	69,154	960.5	348,429	4,839.3	
[12]	Dilithium-III	SW/HW	LW	72	106	40.25	167,433	2,325.5	229,481	3,187.2	634,763	8,816.2	
Security Level 4													
[12]	Dilithium-IV	SW/HW	LW	72	106	40.25	223,272	3,101.0	276,221	3,836.4	815,636	11,328.3	TSMC 40 nm

^{o2} Design for a heuristic parameter set withdrawn by the submitters on Aug. 20, 2019

Table 11: Power and Energy Comparison for Digital Signature Schemes on ASIC

Design	Algorithm	Type	Target	Max. Freq.	Area (kGE)	SRAM (kB)	Key Generation		Signature Verification		Signature Generation		Technology
							Power (mW)	Energy (μJ)	Power (mW)	Energy (μJ)	Power (mW)	Energy (μJ)	
						Security Level 1 & 2							
[12]	qTESLA-I [o2]	SW/HW	LW	72	106	40.25	7.89	531.55	7.99	4.32	9.99	23.34	TSMC 40 nm
[12]	Dilithium-I	SW/HW	LW	72	106	40.25	6.82	9.00	7.73	15.31	6.77	35.41	
[12]	Dilithium-II	SW/HW	LW	72	106	40.25	7.24	13.08	7.49	19.23	7.68	54.82	
						Security Level 3							
[12]	qTesla-III-speed [o2]	SW/HW	LW	72	106	40.25	7.64	1,262.39	7.30	6.86	9.97	43.91	TSMC 40 nm
[12]	qTesla-III-size [o2]	SW/HW	LW	72	106	40.25	7.71	1,229.18	7.59	7.27	9.97	48.23	
[12]	Dilithium-III	SW/HW	LW	72	106	40.25	7.36	17.11	7.41	23.63	7.40	65.26	
						Security Level 4							
[12]	Dilithium-IV	SW/HW	LW	72	106	40.25	6.89	21.38	7.44	28.55	6.93	78.53	TSMC 40 nm

[o2] Design for a heuristic parameter set withdrawn by the submitters on Aug. 20, 2019

All SW/HW co-designs using RISC-V RV32IM

In the study reported in [79], for the most advanced TSMC 28nm library, the ranking of candidates in terms of the most time-critical decapsulation time is *1.* Kyber, *2.* NewHope, *3.* LAC. The difference between positions *1* and *2* is about 6% for the security level 1, and 18% for the security level 5. LAC lags behind NewHope by a factor larger than 3. Kyber also uses less energy than NewHope for each major operation. The energy usage for decapsulation is almost identical for Kyber and NewHope. At the security level 1, Kyber uses about 1% less energy, and at the security level 5 about 7% less energy. At the same time, at the security level 1, LAC requires over 3 times more energy.

Finally, in the study reported in [27], for the UMC 65nm library, the ranking of candidates in terms of the decapsulation time is: *1.* Kyber, *2.* NewHope, *3.* Saber. With very similar power consumption reported for all three candidates, the differences in terms of the execution time translate to the similar differences in energy usage. However, as explained before, this study may be unintentionally biased toward candidates that are able to take advantage of NTT. In particular, for decapsulation, the reported speed-up vs. the baseline pure software implementation running on RISC-V is 9.2, 7.4, and 2.3 for NewHope-512, Kyber-512, and LightSaber, respectively. Similarly, at level 5, the corresponding speed-ups are 9.4, 9.3, and 2.3, respectively. The speed-up for NewHope and Kyber came primarily from a dedicated NTT and Modular Arithmetic Unit. It resulted in a decrease in the decapsulation time for NewHope and Kyber by factors ranging between 2.7 and 3.6 as compared to the best Cortex-M4 software implementation reported to date. The speed-up for Saber came primarily from a half-word integer multiplication and accumulation block.

It resulted in the optimized software/hardware implementations of Saber performing decapsulation in approximately the same or slightly longer execution time than the best Cortex-M4 implementations. As a result, it is reasonable to conclude that additional optimizations might be possible for Saber, bringing its execution time and energy usage to the same range as those of NewHope and Kyber. However, more work is required to demonstrate these improvements in practice. The differences between Kyber and NewHope in terms of the execution times for encapsulation and decapsulation remained at or below 10% for both security levels.

In Table 8, we compare results reported by our own group at the end of 2019 in [19], with results reported by other groups for Saber and NewHope, respectively. All results were obtained using the same SoC FPGA, Zynq UltraScale+. The software/hardware implementation of Round5 was very close to the pure hardware implementation. The same was not the case for the software/hardware implementation of Saber, were a significant percentage of the execution time was devoted to functions remaining in software and to the transfer of data and control between software and hardware. As a result, the most accurate comparison between Round5 and Saber is possible at the security level 3, for which the pure hardware implementation of Saber was reported in [62]. Based on this implementation Saber outperforms Round5 by a small margin in terms of the execution times for encapsulation and decapsulation. At the same time, even the fastest reported implementation of Saber uses 1.6x fewer LUTs than Round5, with the same number of BRAMs and DSP units. FrodoKEM is demonstrated to be by far slower than Saber and Round5 for all security levels.

Somewhat differently, for the security level 5, the pure hardware implementation of NewHope, reported in [41], is not fast enough to outperform the software/hardware implementation of Round5 from [19]. However, the comparison is somewhat complicated by the fact that, in [41], the results are reported the IND-CPA-secure PKE (rather than the IND-CCA-secure KEM), and only the sum of the key generation and decryption (rather than the decryption itself) is reported in the paper.

In Tables 9, 10, and 11, we summarize results available for the implementations of digital signatures. The implementations targeting FPGAs are considered first in Table 9. Unfortunately, multiple results available for qTESLA concern heuristic parameter sets that have been withdrawn by submitters on Aug. 20, 2019. Among the remaining designs, for Artix-7, the ranking of candidates for the security level 1 is *1.* Picnic, *2.* Dilithium, and *3.* qTESLA. The differences among these candidates in terms of the execution time for the signature generation (more critical) and signature verification are very significant. At the same time, only the implementation of Picnic is a high-speed and pure hardware implementation. The remaining implementations are software/hardware implementations based on RISC-V. Additionally, the number of LUTs for Picnic is approximately 6 times larger than for Dilithium, and the number of BRAMs, 3.75 times larger. At the same time, compared to Picnic, the execution time for signature generation is 12 times longer for Dilithium-I and 16 times longer for Dilithium-II.

For security level 3, no implementation of Picnic is available. The implementations of Dilithium-III and qTESLA-p-III are comparable in terms of type, target, and resource utilization. At the same time, the implementation of Dilithium is an order of magnitude more efficient. The implementations of digital signature schemes targeting Kintex-7 and Virtex-7 are summarized in the same table. For the Kintex-7 implementations, Rainbow substantially outperforms Picnic at the security level 1. For all remaining families and security levels, only one candidate with the up-to-date parameter set is reported.

In Tables 10 and 11, Dilithium and qTESLA are compared from the point of view of their execution time, energy usage, and power consumption in ASICs. Unfortunately, the practical importance of the underlying study, reported in [12] and performed in the first half of 2019, was diminished by the use of heuristic parameter sets of qTESLA, withdrawn by the submitters on Aug. 20, 2019.

3 Choice of Algorithms to Implement

In this paper, we focus on KEMs with indistinguishability under chosen-ciphertext attack (IND-CCA). Our primary goal was to implement all lattice-based IND-CCA secure KEMs described in the specifications of Round 2 PQC candidates. Eventually, we fell short of this goal by not implementing a KEM of a single lattice-based candidate, Three Bears. Additionally, we focused on Ring Learning with Rounding (RLWR) variants of Round5, and thus, we did not attempt to implement any LWR variants of this submission.

The submission packages of four candidates – LAC, NTRU, NTRU Prime, and Round5 – describe two substantially different KEMs each. As a result, we have implemented 12 KEMs representing 8 Round 2 candidates. For each implemented KEM, we generated results for all supported security levels.

With a few exceptions, we did not generate results for the underlying public-key encryption schemes (PKE) or concurrently proposed IND-CPA secure KEMs. The reason for that was a focus on the highest-level schemes, which could be securely used to agree on shared session keys, based on the long-term public-private key pairs valid for an extended period of time. In this scenario, the time of the public-private key-pair generation is non-critical, and the design can focus entirely on minimizing the time of encapsulation and decapsulation.

All implemented PQC candidates can be divided into the following major sub-families, listed below together with their Round 2 representatives:

- LWE : Learning With Errors - FrodoKEM
- RLWE : Ring Learning with Errors - LAC (including LAC-v3a and LAC-v3b) and NewHope
- Module-LWE: Module Learning with Errors - CRYSTALS-KYBER
- RLWR : Ring Learning With Rounding - Round5 (with and without an error correcting code)

Table 12: Features of selected NIST Round 2 PQC KEMs

Feature	**LAC-(v3a/v3b)**	**NewHope**	**Round5**	**Kyber**	**Saber**	**FrodoKEM**
Underlying problem	Ring-LWE: Ring Learning With Errors	Ring-LWE: Ring Learning With Errors	RLWR: Ring Learning With Rounding	Module-LWE: Module Learning with Errors	Mod-LWR: Module Learning with Rounding	LWE: Learning With Errors
Degree n	Power of 2	Power of 2	$2^8 < n < 2^{11}$	Power of 2	Power of 2	$n \equiv 0 \pmod 8$
Modulus q	Byte-level Prime / Power of 2	Prime	Power of 2	Prime	Power of 2	Power of 2
Other major parameters	ψ_n^h: Binomial distribution, $[l_c, l_m, l_d]$: BCH code	k: noise parameter, γ: NTT parameter	p, t: other moduli	k: the lattice dimension as a multiple of n, η: noise parameter	l: number of polynomials per vector, p, T: other moduli, μ: parameter of CBD	B: number of bits, encoded in each matrix entry, σ: standard deviation
Hash-based functions	SHA3-512 SHAKE256	SHAKE128, SHAKE256	1: SHAKE128 3, 5: SHAKE256	SHA3-256, SHA3-512, SHAKE128, SHAKE256	SHA3-256, SHA3-512, SHAKE128	1: SHAKE128 3, 5: SHAKE256
Sampling	Integers are sampled from a fixed-weight centered binomial distribution(CBD)	Integers are sampled from a centered binomial distribution (CBD)	Integers from a uniform distribution are produced by a DRBG taking a random seed	Integers are sampled from a centered binomial distribution (CBD)	Integers are sampled from a centered binomial distribution (CBD)	Integers are sampled from an approximation of a rounded continuous Gaussian distribution
Decryption failures	Yes	Yes	Yes	Yes	Yes	Yes
Polynomial Rings	$\mathbb{Z}_q[x]/(x^n+1)$	$\mathbb{Z}_q[x]/(x^n+1)$	$Z_q[x]/\Phi_{n+1}{}^{**}$	$\mathbb{Z}_q[x]/(x^n+1)$	$\mathbb{Z}_q[x]/(x^n+1)$	None
#Polynomial Multiplications in Encapsulation	2	2	2	k^2+k	l^2+l	None 2 matrix-by-matrix*
#Polynomial Multiplications in Decapsulation	3	3	3	k^2+2k	l^2+2l	None 3 matrix-by-matrix*

* Elements of matrices in Z_q

** $\Phi_{n+1} = (x^{n+1}-1)/(x-1)$

Table 13: Features of NIST Round 2 NTRU-based PQC KEMs

Feature	**NTRU-HPS**	**NTRU-HRSS**	**Streamlined NTRU Prime**	**NTRU LPRime**
Underlying problem	Shortest Vector Problem	Shortest Vector Problem	Shortest Vector Problem	Shortest Vector Problem
Polynomial P	$x^n - 1$	$\Phi_n = (x^n - 1)/(x - 1)^{**}$	$x^n - x - 1$ irreducible in $Z_q[x]$	$x^n - x - 1$ irreducible in $Z_q[x]$
Degree n^*	Prime	Prime	Prime	Prime
Modulus q	power of 2 with $q/8 - 2 \leq 2n/3$	power of 2 with $q > 8\sqrt{2}(n + 1)$	Prime	Prime
Other major parameters	w: Fixed weight for f and r	N/A	w: Fixed weight for f and r. $3w \leq 2n$ $16w + 1 \leq q$	w: Fixed weight for b and a. $3w \leq 2n$ $16w + 2\delta + 3 \leq q$
Hash-based functions	SHA3-256	SHA3-256	SHA3-512	SHA3-512
Sampling	Fixed-weight and variable -weight polynomials are sampled from a uniform distribution	Variable-weight polynomials are sampled from a uniform distribution	Fixed-weight polynomials are sampled from a uniform distribution	Fixed-weight polynomials are sampled from a uniform distribution
Decryption failures	No	No	No	No
Polynomial Rings	R/q: $Z_q[x]/(x^n - 1)$ S/q: $Z_q[x]/(\Phi_n)^{**}$ S/3: $Z_3[x]/(\Phi_n)^{**}$	R/q: $Z_q[x]/(x^n - 1)$ S/3: $Z_3[x](x - 1)/(x^n - 1)$	R/q: $Z_q[x]/(x^n - x - 1)$ R/3: $Z_3[x]/(x^n - x - 1)$	R/q: $Z_q[x]/(x^n - x - 1)$ R/3: $Z_3[x]/(x^n - x - 1)$
#Polynomial Multiplications in Encapsulation	1 in R/q	1 in R/q	1 in R/q	2 in R/q
#Polynomial Multiplications in Decapsulation	1 in R/q 1 in S/q 1 in S/3	1 in R/q 1 in S/q 1 in S/3	2 in R/q 1 in R/3	3 in R/q

* Denoted by p in the specification of Streamlined NTRU Prime and NTRU LPRime

** $\Phi_n = (x^n - 1)/(x - 1)$ irreducible in $Z_q[x]$

- Mod-LWR : Module Learning with Rounding - Saber,
- NTRU-based : NTRU (including NTRU-HPS and NTRU-HRSS) and NTRU Prime (including Streamlined NTRU Prime and NTRU LPRime).

Both implemented variants of LAC were announced in the middle of Round 2, on Dec. 19, 2019. The implemented variants of the remaining algorithms have remained unchanged since the beginning of Round 2.

The following two submissions did not limit the generation of pseudorandom bits to any particular algorithm (e.g., SHAKE): LAC and NTRU. As a result, for each of them, we selected a variant of a pseudorandom number generator most efficient on our benchmarking platform. In the case of CRYSTALS-Kyber, we selected one of the variants described in the specification - a variant based on the SHA-3 functions.

Selected features of all implemented KEMs are summarized in Tables 12 and 13.

In all of these KEMs, the elementary operation is multiplication mod q. In FrodoKEM, LAC-v3b, Round5, Saber, NTRU-HPS, and NTRU-HRSS, q is a power of two, which significantly simplifies the reduction mod q. In NewHope and Kyber, q is a special prime, selected in such a way to support speeding up polynomial multiplication in $\mathbb{Z}_q[x]/(x^n + 1)$ using the Number Theoretic Transform (NTT). In LAC-v3a, q is a one-byte prime (251). In Streamlined NTRU Prime and NTRU LPRime, it is a prime smaller than 2^{13}. The moduli chosen for NTRU Prime algorithms may potentially lead to a higher resistance against future attacks.

In FrodoKEM, the most time-consuming operation is a matrix-by-matrix multiplication, where each component of a matrix is an element of Z_q. In Kyber and Saber, the most

Table 14: Parameter sets of investigated algorithms. Notation: Sk - Secret Key, Pk - Public key, Ct - Ciphertext.

Algorithm	Parameter Set	Security Level	Degree n	Modulus q	Sk Size [bytes]	Pk Size [bytes]	Ct Size [bytes]
FrodoKEM	Frodo-640	1	640	2^{15}	19,888	9,616	9,720
Kyber	KYBER512	1	256	3329	1,632	800	736
LAC-v3a	LAC-128	1	512	251	1,056	544	704
LAC-v3b	LAC-128	1	512	256	1,056	544	704
NewHope	NEWHOPE512-CCA-KEM	1	512	12289	1,888	928	1,120
NTRU-HPS	ntruhps2048677	1*	77	2^{11}	1,235	931	931
NTRU-HRSS	ntruhrss701	1*	701	2^{13}	1,452	1,138	1,138
Str NTRU Prime	kem/sntrup653	2	653	$4621 < 2^{13}$	1,518	994	897
NTRU LPRime	kem/ntrulpr653	2	653	$4621 < 2^{13}$	1,125	897	1,025
Round5	R5ND_CCA_1KEM_0d	1	586	2^{13}	708	676	740
Round5	R5ND_CCA_1KEM_5d	1	508	2^{10}	493	461	620
Saber	LightSaber-KEM	1	256	2^{13}	1,568	672	736
FrodoKEM	Frodo-976	3	976	2^{16}	31,296	15,632	15,744
Kyber	KYBER768	3	256	3329	2,400	1,184	1,088
LAC-v3a	LAC-192	3	1024	251	2,080	1,056	1,352
LAC-v3b	LAC-192	3	1024	256	2,080	1,056	1,352
NTRU-HPS	ntruhps4096821	3*	821	2^{12}	1,592	1,230	1,230
Str NTRU Prime	kem/sntrup761	3	761	$4591 < 2^{13}$	1,763	1,158	1,039
NTRU LPRime	kem/ntrulpr761	3	761	$4591 < 2^{13}$	1,294	1,039	1,167
Round5	R5ND_CCA_3KEM_0d	3	852	2^{12}	1,031	983	1,103
Round5	R5ND_CCA_3KEM_5d	3	756	2^{12}	828	780	934
Saber	Saber-KEM	3	256	2^{13}	2,304	992	1,088
Str NTRU Prime	kem/sntrup857	4	857	$5167 < 2^{13}$	1,463	1,184	1,312
NTRU LPRime	kem/ntrulpr857	4	857	$5167 < 2^{13}$	1,999	1,322	1,184
FrodoKEM	Frodo-1344	5	1344	2^{16}	43,088	21,520	21,632
Kyber	KYBER1024	5	256	3329	3,168	1,568	1,568
LAC-v3a	LAC-256	5	1024	251	2,080	1,056	1,464
LAC-v3b	LAC-256	5	1024	256	2,080	1,056	1,464
NewHope	NEWHOPE1024-CCA-KEM	5	1024	12289	3,680	1,824	2,208
Round5	R5ND_CCA_5KEM_0d	5	1170	2^{13}	1,413	1,349	1,509
Round5	R5ND_CCA_5KEM_5d	5	946	2^{11}	1,042	978	1,285
Saber	FireSaber-KEM	5	256	2^{13}	3,040	1,312	1,472

* assuming non-local computational models

time-consuming operations are matrix-by-vector and vector-by-vector multiplications, where each element of a matrix or a vector is a polynomial with n coefficients in Z_q, and the multiplication of such polynomials is performed modulo the reduction polynomial $x^n + 1$. In New Hope, LAC, Round5, and all NTRU-based KEMs, the most time-consuming operation is a polynomial multiplication.

The only KEMs with no Decryption Failure in the underlying PKE are NTRU-based KEMs (NTRU-HPS, NTRU-HRSS, Streamlined NTRU Prime, and NTRU LPRime).

Round5 and NTRU-based KEMs use sampling from the uniform distribution. In LAC, NewHope, Kyber, and Saber, a Centered Binomial Distribution (CBD) is used. In FrodoKEM, an approximation of a rounded continuous Gaussian distribution is required.

Parameter sets of 12 investigated algorithms are summarized in Table 14. The specification of NTRU associates two different security categories with each parameter set for NTRU-

HPS and NTRU-HRSS. In this paper, we conservatively assumed the lower security level based on the so-called non-local computational models (see [71], Section 5.3 Security Categories). The same computation model is implicitly assumed by the submitters of the other investigated algorithms.

In Table 14, we have divided parameter sets into three groups with security levels 1 and 2, 3 only, and 4 and 5, respectively. Only the first group contains variants of all 12 investigated algorithms (with ten at level 1 and two at level 2). The second group includes 10 variants at the security level 3. Finally, the last group includes 10 variants total (with two at level 4 and eight at level 5).

4 Methodology

4.1 Assumptions

All implemented schemes are Key Encapsulation Mechanisms (KEMs). For each of them, we support two major operations: Encapsulation and Decapsulation. Whenever possible, hardware resources and software functions are shared between these two operations. All parameter sets of the given PQC scheme share the same HDL code. At the same time, the choice among parameter sets is made at the time of synthesis, so the exact amount of FPGA resources required to implement each particular parameter set can be determined and reported. The key generation is assumed to be performed in software or using a separate hardware unit.

Based on the considerations discussed in Section 1, our optimization target is high-speed for both hardware and software/hardware implementation approaches. In both cases, the primary goal is the minimum execution time for Encapsulation and Decapsulation. No explicit limits are imposed on any resources of the FPGA platform, such as Configurable Logic Block Slices, LUTs, flip-flops, BRAMs, or DSP units. The goal is to demonstrate each algorithm's inherent ability to execute multiple operations in parallel.

All implementations are required to be constant-time to make them resistant against any known timing attacks. No physical access to the device or its proximity is assumed, which means that countermeasures against power-based and electromagnetic analysis-based attacks are considered non-essential. Developing and implementing such countermeasures is beyond the scope of this study.

HDL code is required to be portable among multiple state-of-the-art FPGA families of Xilinx and Intel, assuming that a given design fits in the largest device of a given family. The code does not use any vendor or family-specific primitives or megafunctions. Each hardware unit uses only a single clock. This clock can operate at an arbitrary clock frequency lower than or equal to the maximum clock frequency determined by the critical path of a given hardware unit. All reported execution times correspond to this maximum clock frequency.

4.2 Choice of Benchmarking Platforms for Round 2

Hardware. The submissions selected for the hardware-only implementations (CRYSTALS-KYBER, LAC, New Hope, and Round5) have moderate resource requirements, even when optimized for high-speed. As a result, we have decided to generate results for two FPGA families: Artix-7 and Virtex-7. Based on Section 2, these families were selected for benchmarking by the largest number of other groups to date.

Software/hardware co-design. In recent years, several hardware/software co-design platforms have emerged. The most popular in the industry are those based on integrating an ARM-based processor and FPGA fabric on a single chip. Examples include Xilinx Zynq 7000 System on Chip (SoC), Xilinx Zynq UltraScale+ MPSoC, Intel Cyclone V SoC

FPGAs, Intel Arria 10 SoC FPGAs, and Intel Agilex F-Series SoC FPGAs. These devices support software/hardware co-designs based on a traditional high-level language program running on an ARM processor, with the most time-critical computations performed on a dedicated hardware accelerator. The advantages of these platforms include the use of the most popular embedded processor family (ARM) operating at high speed (1 GHz or above), state-of-the-art commercial tools (available for free, or at a reduced price for academic use), availability of relatively inexpensive prototyping boards, and practical deployment in multiple environments.

The primary alternatives are FPGA-based systems with so-called "soft" processor cores implemented in reconfigurable logic. Examples include Xilinx MicroBlaze, Intel Nios II, and the open-source RISC-V, originally developed at the University of California, Berkeley [59, 77, 78]. The main advantage of these systems over "hard" processor cores is flexibility in the allocation of resources to processor cores, including the possibility of extending them with special instructions specific to PQC. Additionally, they are easy to port between different FPGA families, and even between FPGAs and ASICs. A disadvantage compared to the "hard" option is that the "soft" processors operate at much lower clock frequencies (typically 200-450 MHz).

During Round 2, NIST asked designers to focus on the ARM Cortex-M4 for embedded software implementations and the Artix-7 for FPGA implementations. However, we are not aware of any SoC FPGA that contains a Cortex-M processor and the Artix-7 FPGA fabric on a single chip. Even if such a chip existed, it would be more suitable for benchmarking of lightweight implementations (optimized for minimum cost and power consumption), rather than benchmarking of the high-speed implementations targeted by our study.

As a result, we have based our choice of a platform primarily on the projected practical importance of various platforms during the initial period of deploying new PQC standards, and the expected speed-up over pure-software implementations. These priorities led us to choose devices from the "hard" processor class, with a hard-wired ARM processor, and among them, the Zynq UltraScale+ family from Xilinx Inc., the vendor with the biggest market share in this device category. Zynq UltraScale+ and similar SoC FPGAs are likely to be used for practical deployments of PQC in the near future, wherever device speed and time-to-market are of primary concern. Implementations using these devices are even more likely than implementations using only hardware.

However, the use of soft-core processors, and in particular the free and open-source RISC-V, should be considered as a natural next step, especially in light of DARPA's recent selection of the RISC-V Instruction Set Architecture (ISA) for investigation within its cybersecurity-related programs [55]. Since these soft-core processors can be implemented practically on any modern FPGA family, the choice of the family should be dependent primarily on the selected type of implementation: lightweight vs. high-speed.

Based on the above discussion, we chose the Xilinx Zynq UltraScale+ MPSoC XCZU9EG-2FFVB1156E as our target device and the Xilinx ZCU102 Evaluation Kit as a prototyping board.

Our target device, Xilinx Zynq UltraScale+ MPSoC XCZU9EG-2FFVB1156E, is composed of two major parts sharing the same chip. The primary component of the Processing System (PS) is a quad-core ARM Cortex-A53 Application Processing Unit, running at 1.2 GHz. As in the software benchmarking experiments conducted by other groups, we utilize only one core in all our experiments. The Programmable Logic (PL) includes a programmable FPGA fabric similar to that of Virtex UltraScale+ FPGAs, including Configurable Logic Block (CLB) slices, Block RAMs, DSP units, etc. The frequency of operation depends on the particular logic instantiated in the reconfigurable fabric but typically does not exceed 400 MHz.

Computer-Aided Design Tools. The software used is Xilinx Vivado Design Suite HLx Edition, and Xilinx Software Development Kit (XSDK), all with version number 2018.2.

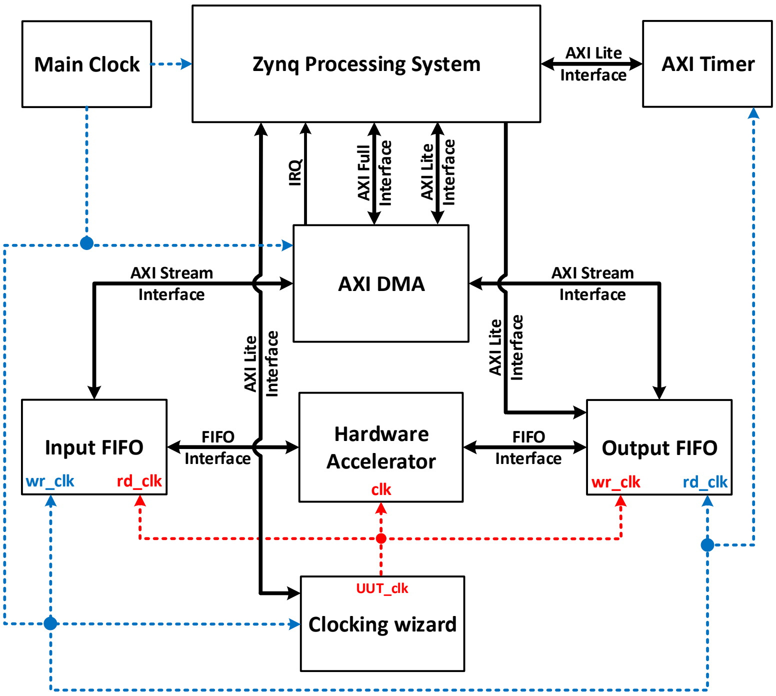


Figure 1: Block diagram of software/hardware co-design.

4.3 Benchmarking Setup for Software/Hardware Co-design

A high-level block diagram of the experimental software/hardware co-design platform is shown in Fig. 1. The Hardware Accelerator is connected, through the dual-clock Input and Output FIFOs, to the AXI DMA, supporting the high-speed communication with the Processing System. Timing measurements are performed using the popular Xilinx IP unit called AXI Timer, which is capable of measuring time in clock cycles of the 200 MHz system clock. The Hardware Accelerator can operate at a variable clock frequency, controlled from software using the Clocking wizard unit.

4.4 Interface and Communication Protocol

The interface of the hardware accelerator is shown in Fig. 2. This interface is assumed to be identical for both hardware and software/hardware implementations and matches the interface of the Input and Output FIFOs, shown in Fig. 3. The default width of the data bus is 64 bits. Each particular operation, such as load public key, start encapsulation, etc., is initiated by sending an appropriate header (in the form of a single 64-bit word) from a program running on the ARM processor to the data input of a hardware accelerator. When an operation requires additional data, this data is transmitted using the subsequent Input FIFO words.

After the hardware accelerator produces results or detects an error, a header word is sent in the opposite direction. If an additional output is required, this output follows the header and is arranged in 64-bit words. The detailed format of the exchanged inputs and outputs is left up to the designer of a hardware accelerator.

Compared to an earlier proposed PQC Hardware API [26], the adopted interface is significantly simpler and more flexible. Only one input port, infifo, is used in place of three separate ports, Public Data Input (PDI), Secret Data Input (SDI), and Random Data Input (RDI). Only one output port, outfifo, is used in place of two separate ports,

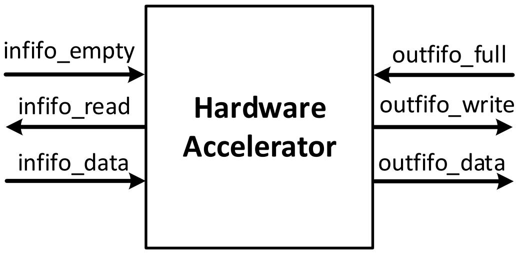
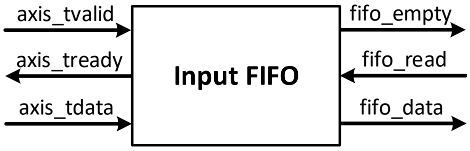
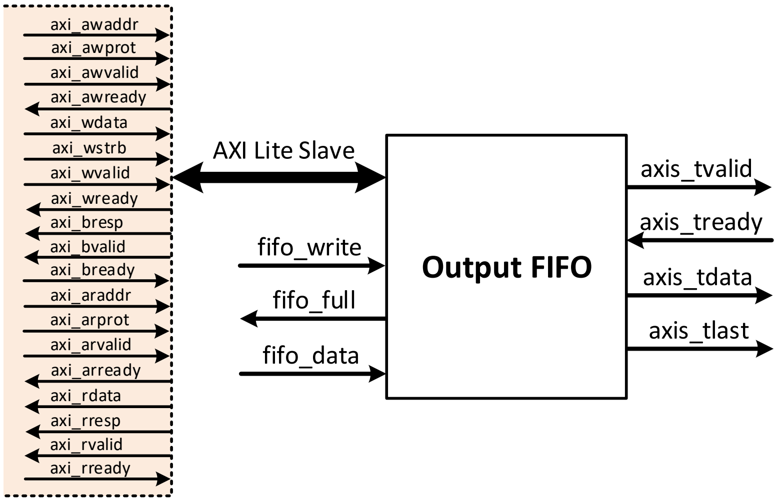


Figure 2: Hardware accelerator interface.



(a) Input FIFO



(b) Output FIFO

Figure 3: The Input and Output FIFO Interface.

Public Data Output (PDO) and Secret Data Output (SDO).

The proposed interface does not provide physical separation among the public, secret, and random data. Still, it appears to be sufficient at the current stage of the evaluation process for both pure hardware and software/hardware implementations. It also significantly simplifies the software/hardware partitioning and transfer of data between the processor and the hardware accelerator.

4.5 Porting Software Implementations to ARM Cortex-A53

To minimize overhead, we have run software in the Bare Metal mode, without any operating system. We have started from the best high-level language implementations of selected candidates available to date. In order to be run on ARM Cortex-A53 in the Bare Metal mode, these implementations had to be modified as described below.

Since no functions of Open-SSL are available in the Bare Metal mode, we have adopted for AES the Optimized ANSI C code of the Rijndael cipher based on the use of T-boxes, developed by Vincent Rijmen, Antoon Bosselaers, and Paulo Barreto [61]. Compared to the OpenSSL implementation, the selected implementation is written entirely in C, rather than in an assembly language of a specific processor. It does not contain any countermeasures against cache-timing attacks.

For SHA-3, for all candidates other than Round5, we adopted fips202.c from SUPERCOP by Ronny Van Keer, Gilles Van Assche, Daniel J. Bernstein, and Peter Schwabe. For Round5, we used r5_xof_shake.c by Markku-Juhani O.Saarinen and keccak1600.c from SUPERCOP, by the same authors as fips202.c

For all investigated KEMs, the encapsulation operation uses multiple calls to the function `randombytes()`, which produces a sequence of random bytes with uniform distribution. Other PQC benchmarking projects use a version of this function based on operating system functions and/or functions from OpenSSL [15, 63, 43, 67]. None of these options is available in the Bare Metal mode. Therefore, in our code, we use the implementation of `randombytes()` proposed by Saarinen in April 2018 [63], which is an improved version of the implementation developed by NIST for the generation of known-answer tests [58]. Since both of these implementations are based on AES in the ECB mode, from the OpenSSL library, we have replaced the code of AES by the mentioned above standalone, optimized implementation of AES in C [61]. As a result, the selected implementation of `randombytes()` is likely to have different timing characteristics than the implementations used in other benchmarking studies, such as SUPERCOP [15], pqcbench [63], pqm4 [43], and liboqs [67].

Taking into account that the C implementations of NTRU-HPS, NTRU-HRSS, and Streamlined NTRU Prime use `randombytes()` to generate 3211, 1400, and 2611 bytes, respectively, we have sped up these function calls by a) limiting the number of bytes returned by `randombytes()` to 32, and b) generating the remaining random bytes using SHAKE128. For the three KEMs mentioned above, this change resulted in the speed-up of the relevant functions by a factor greater than 3.

No attempt at the optimization of the software implementations of KEMs by employing assembly language coding has been made.

4.6 Software Profiling, C Source Code Analysis, and Software/Hardware Partitioning

Our first step in evaluating the suitability of cryptographic algorithms for software/hardware co-design was profiling of their software implementations using one core of the ARM Cortex-A53. Profiling produced a list of the most time-consuming functions, including their absolute execution time, percentage execution time, and the number of times they are called.

We decided which functions to offload to hardware based on the highest potential for the total speed-up, as well as the fairness of comparison among investigated algorithms. The total speed-up obtained by offloading an operation to hardware depends on two major factors: the percentage of the execution time taken in software by an operation offloaded to hardware, and the speed-up for the offloaded operation itself. In order to maximize the first factor, we gave priority to operations that take the largest percentage of the execution time, preferably more than 90%. These operations may involve a single function call, several adjacent function calls, or a sequence of consecutive instructions in C. It is preferred that a given operation is executed only once, or only a few times, as each transfer of control and data between software and hardware involves a certain fixed timing overhead, independent of the size of input and output to the accelerator. In order to maximize the second factor, we gave priority to operations that have a high potential for parallelization in hardware, and a small total size of inputs and outputs (which will need to be transferred to and from the hardware accelerator, respectively)

Most of the data required to make informed decisions regarding software/hardware partitioning can be obtained by profiling software implementations, possibly extended with some small modifications required to gather all relevant data. However, determining the potential for parallelization requires some knowledge of hardware or at least basic concepts of concurrent computing.

To assure fairness in our comparison, we offloaded to hardware all operations common to or similar across the implemented algorithms (e.g., all polynomial multiplications), and all operations that contributed significantly to the total execution time. Nevertheless, it should be understood that this heuristic procedure may need to be repeated several times because, after each round of offloading to hardware, different software operations may emerge as taking the majority of the total execution time. This process can stop when the development effort required for offloading the next most-critical operation to hardware is disproportionately high compared to the projected speed-up.

All encapsulations involve a single call to the function `randombytes()`, returning a seed to a pseudorandom number generator. This function could be possibly offloaded to hardware by implementing a True Random Number Generator (TRNG) in Programmable Logic. However, the correct implementation of a TRNG in FPGA fabric is a substantial project by itself. Additionally, in some cases, the seed would need to be transferred back to software, while in others it could be used directly by a hardware accelerator. To avoid these additional complications, in all the current software/hardware partitioning schemes, the seed is assumed to be generated in software.

4.7 RTL Design Methodology

The design of a hardware accelerator follows a traditional Register-Transfer Level (RTL) methodology. The entire system is divided into the Datapath and Controller. The Datapath is described using a hierarchical block diagram, and the Controller using hierarchical algorithmic state machine (ASM) charts. Multiple local controllers may be advantageous compared to a single global Controller. The RTL approach, although not novel by itself, is an important part of our methodology as it facilitates very efficient hardware accelerator designs. The block diagrams and ASM charts are very easy to translate to efficient and fully synthesizable VHDL code.

4.8 Potential Software Optimizations

Our software/hardware implementations could be potentially sped up by accelerating the remaining software part using assembly language programming.

The ARM Cortex-A53 is a microarchitecture implementing the ARMv8-A 64-bit instruction set. This instruction set consists of traditional RISC instructions operating on 31 general-purpose 64-bit registers, as well as Single-Instruction Multiple-Data (SIMD) instructions operating on 32 128-bit registers, treated as vectors composed of smaller data words. The SIMD architecture extension for the ARM Cortex-A processors, including Cortex-A53, is referred to as NEON [7]. The NEON 128-bit registers are considered as vectors of elements of the same data type, with NEON instructions operating on multiple elements simultaneously. Multiple data types are supported by this technology, including floating-point and integer operations.

A programmer can take advantage of NEON instructions using any of the following methods: a) Auto-vectorization by a compiler, b) Use of NEON-enabled libraries, c) using NEON intrinsics, and d) Using hand-coded NEON assembly language code [7].

Auto-vectorization is the process by which a compiler automatically analyzes the code and identifies opportunities to optimize performance using NEON instructions and register files. NEON-enabled libraries focus on signal and image processing, computer vision, physics, and machine learning. Examples include Arm Compute Library, Ne10, Libyuv, and Skia. NEON intrinsics are function calls that the compiler replaces with an appropriate NEON instruction or sequence of NEON instructions. Intrinsics provide almost as much control as writing assembly language but leave the allocation of registers to the compiler [6]. Finally, using hand-coded NEON assembly language code provides the programmer with the highest level of control, which may be used for the most aggressive and advanced optimizations.

Our implementations take advantage of only Option a) Auto-vectorization by a compiler. Regarding Option b), we are not aware of any NEON-enabled library that explicitly benefits lattice-based cryptosystems. No attempt was made to optimize software parts of our implementations using either intrinsics or hand-coded NEON assembly.

Our justification for these choices is as follows. Operations that are most suitable for a speed-up using NEON instructions are typically also excellent candidates for offloading to hardware. As a result, at the time when further offloading to hardware is judged to be either counterproductive or too labor intensive, the remaining operations executed in software are most likely sequential in nature and cannot take advantage of NEON instructions and registers.

Still, some speed up could be potentially accomplished by hand-coding these operations using scalar instructions of the ARMv8-A 64-bit instruction set. However, doing that would make the implementation much less portable. A similar effort could possibly be better spent on offloading all remaining operations to hardware. Even if the further offloaded operations cannot by themselves benefit from any substantial speed-up, moving them to hardware will eventually eliminate the entire transfer time, which remains substantial in all our software/hardware co-designs.

As a result, optimizing software implementations using NEON instructions and registers should be treated as an alternative optimization path, starting from the same starting point as our software/hardware co-designs. This starting point is a portable C implementation, that can be easily profiled and analyzed for inherent parallelism.

4.9 Verification and Generation of Results

Functional verification of the hardware description language (HDL) code is performed by comparing simulation results with precomputed outputs generated by a reference software implementation.

Fully verified and independently optimized VHDL code is then combined with the *optimized* software implementation of a given PQC candidate. Functional verification of the integrated software/hardware design is performed by running the code on the prototyping board and comparing the obtained outputs with outputs generated by a functionally equivalent reference implementation, run on the same ARM Cortex-A53 processor.

Experimental timing measurements follow, with the hardware accelerator's clock set (using the Clocking wizard) to the optimal target frequency identified during the synthesis and implementation runs. The execution time is measured by using the AXI Timer module, shown in Fig. 1, in clock cycles of the AXI Timer, which operates at the default clock frequency of 200 MHz.

The encapsulation time does not include the time necessary to transfer public key to the hardware accelerator. Similarly, the decapsulation time does not include the time necessary to transfer private key to the hardware accelerator. In case a public key of the receiver is required during decapsulation, this key is assumed to be a part of the corresponding private key.

The time required for a key upload is calculated as a difference between the time necessary for transferring a concatenation of the key and the first input (e.g., the seed for encapsulation, or the ciphertext for decapsulation) minus the time required to transfer the first input itself. This convention is consistent with the fact that the transmission of the key does not need to be repeated if the same key is reused multiple times. At the same time, the key upload overhead is typically so small that it is not efficient to send the key well before its first use. As a result, the key upload is assumed to be always combined with the transmission of the first input.

5 Results

5.1 Results for Hardware Implementations

Table 15: Maximum frequency and resource utilization of hardware implementations on Artix-7.

Algorithm	**Security Category: Parameter Set**	**Max. Freq.**	**LUT**	**FF**	**Slice**	**DSP**	**BR AM**
Kyber	1: KYBER512	210	11,864	10,348	3,989	8	15.0
Kyber	2: KYBER768	210	11,884	10,380	3,984	8	15.0
Kyber	5: KYBER1024	210	12,183	12,441	4,511	8	15.0
LAC-v3a	1: LAC-128	185	23,314	15,950	7,099	0	8.5
LAC-v3a	3: LAC-192	172	38,898	26,174	11,700	0	11.5
LAC-v3a	5: LAC-256	167	42,721	26,872	12,903	0	11.5
LAC-v3b	1: LAC-128	192	18,955	15,958	5,421	0	8.5
LAC-v3b	3: LAC-192	190	28,362	26,182	7,949	0	11.5
LAC-v3b	5: LAC-256	167	32,184	26,882	8,995	0	11.5
NewHope	1: NEWHOPE512-CCA-KEM	225	9,000	8,732	3,194	4	12.0
NewHope	5: NEWHOPE1024-CCA-KEM	225	9,000	8,732	3,194	4	12.0
Round5	1: R5ND_CCA_1KEM_0d	185	57,137	80,676	21,291	0	3.0
Round5	3: R5ND_CCA_3KEM_0d	165	78,825	107,564	29,441	0	3.0
Round5	5: R5ND_CCA_5KEM_0d			Doesn't fit			
Round5	1: R5ND_CCA_1KEM_5d	204	36,578	56,355	14,042	0	3.0
Round5	3: R5ND_CCA_3KEM_5d	174	59,852	95,170	24,869	0	3.0
Round5	5: R5ND_CCA_5KEM_5d	169	69,548	113,913	28,286	0	3.0

Table 16: Maximum frequency and resource utilization of hardware implementations on Virtex-7.

Algorithm	**Security Category: Parameter Set**	**Max. Freq.**	**LUT**	**FF**	**Slice**	**DSP**	**BR AM**
Kyber	1: KYBER512	245	13,745	11,107	4,590	8	14.0
Kyber	3: KYBER768	245	13,889	11,113	4,500	8	14.0
Kyber	5: KYBER1024	245	14,163	13,179	5,172	8	14.0
LAC-v3a	1: LAC-128	286	24,452	16,097	7,320	0	8.5
LAC-v3a	3: LAC-192	250	39,220	26,325	12,021	0	11.5
LAC-v3a	5: LAC-256	208	44,722	27,033	13,659	0	11.5
LAC-v3b	1: LAC-128	294	18,972	16,061	5,300	0	8.5
LAC-v3b	3: LAC-192	286	28,344	26,206	7,654	0	11.5
LAC-v3b	5: LAC-256	213	32,177	26,846	9,111	0	11.5
NewHope	1: NEWHOPE512-CCA-KEM	295	11,345	8,838	3,572	4	12.0
NewHope	5: NEWHOPE1024-CCA-KEM	295	11,345	8,838	3,572	4	12.0
Round5	1: R5ND_CCA_1KEM_0d	238	62,407	80,726	23,918	0	3.0
Round5	3: R5ND_CCA_3KEM_0d	208	78,727	107,631	28,034	0	3.0
Round5	5: R5ND_CCA_5KEM_0d	215	108,472	156,532	39,008	0	3.0
Round5	1: R5ND_CCA_1KEM_5d	256	38,350	56,413	14,731	0	3.0
Round5	3: R5ND_CCA_3KEM_5d	222	59,824	95,270	23,505	0	3.0
Round5	5: R5ND_CCA_5KEM_5d	202	69,561	113,933	28,643	0	3.0

Six CCA-secure KEMs representing four candidate - CRYSTALS-Kyber, LAC, NewHope, and Round5 - have been implemented in pure hardware. LAC is represented by two variants, v3a with q=251 and v3b with q=256. Round5 is represented by R5ND_CCA_KEM_0d

- a ring variant without any error correcting code, and R5ND_CCA_KEM_5d - a ring variant with the XE5 forward error correcting code used to decrease decryption failure rates during decapsulation (and thus improve bandwidth and security).

The maximum clock frequency and resource utilization of our hardware implementations of all six KEMs are summarized in Table 15 for Xilinx Artix-7 FPGAs and in Table 16 for Xilinx Virtex-7 FPGAs. All but one KEM fit within the largest device of the Artix-7 family. The only one that does not is the security level 5 variant of Round5 without error correction. Taking into account that we target high-speed implementations, more suitable for high-performance FPGAs, such as Virtex-7, the inability to fit the high-speed version of the highest-security variant in low-cost FPGA family should not be used against Round5.

LAC-v3b, with q=256, clearly outperforms LAC-v3a, with q=251, in terms of both the maximum clock frequency and resource utilization. For example, for the security level 1 on Artix-7, the implementation of LAC-v3b has about 4% higher frequency and requires about 19% fewer LUTs. In the case of Round5, R5ND_CCA_KEM_5d (with error correction) significantly outperforms R5ND_CCA_KEM_0d (without error correction). For example, at the security level 1 on Artix-7, the difference is at the level of 10% in terms of clock frequency, and 36% in terms of the number of LUTs.

Taking into account the best variants of all four submissions at the security level 1, all clock frequencies are in a very small range between 192 and 210 MHz for Artix-7 and between 235 and 294 for Virtex-7. Thus, no significant advantage in terms of the maximum clock frequency is demonstrated by any candidate.

Ranking of candidates in terms of resource utilization is also very difficult because of no clear equivalence between various elements of the resource utilization vectors. For example, on Artix-7, NEWHOPE512-CCA-KEM uses about 4 times fewer LUTs than R5ND_CCA_1KEM_5d, but requires 4 vs. 0 DSP units, and 4 times more BRAMs. Thus, none of these implementations can be claimed to be clearly superior vs. the other.

However, an important differentiating factor is the use of either similar or significantly different amount of resources for implementing different security levels. It is generally more desirable to have an algorithm that can be implemented using the same amount of resources, independently of the security level. This feature allows an easier upgrade of a security level. It also indirectly implies that the 3-in-1 or 2-in-1 designs will have a similar resource utilization as the lowest-security variant rather than the resource utilization higher than that of the highest-security variant. Out of six investigated KEMs, this desirable property is exhibited only by Kyber and NewHope. On top of that, Kyber is slightly more flexible, due to the existence of a variant at the security level 3. On the other hand, NewHope has a small advantage in terms of all elements of the resource utilization vector (e.g., for level 1 at Artix-7, it uses 9000 vs. 11,864 LUTs, 8,732 vs. 10,348 FFs, 3,194 vs. 3,989 slices, 4 vs. 8 DSP units, and 12 vs. 15 BRAMs).

The times necessary to load a public key (required for encapsulation) and a secret (private) key (required for decapsulation) are proportional to the size of the respective key and inversely proportional to the maximum clock frequency of a given PQC unit. All transfers are assumed to be conducted using a 64-bit infifo_data bus. The sizes of keys for all variants of all investigated algorithms are summarized in Table 14. The maximum clock frequencies are listed in Table 15 for Artix-7 and Table 16 for Virtex-7. In Fig. 4, we compare these key loading times for Artix-7, and in Fig. 5 for Virtex-7. For both Artix-7 and Virtex-7, R5ND_CCA_KEM_5d has the shortest key-loading times, and NewHope the longest. However, the differences among these times are relatively minor. They do not exceed a factor of 2 for loading a public key, and 3 for loading a private key.

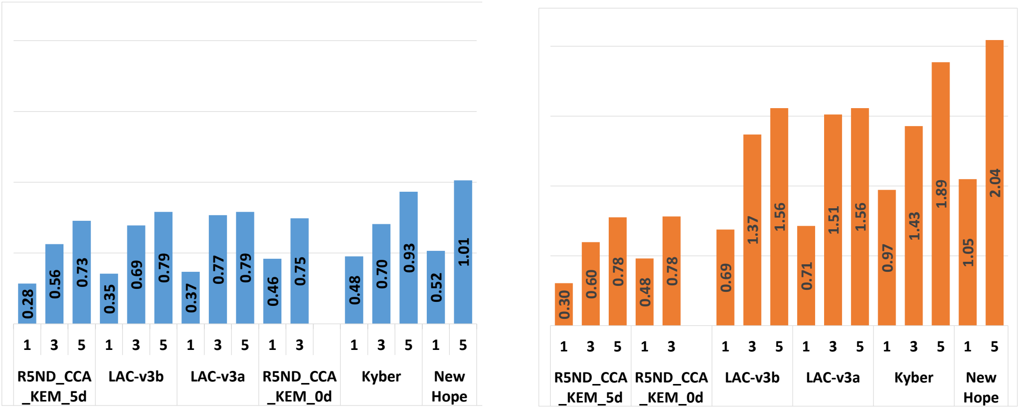
The ranking of all 6 implemented KEMs in terms of the two primary performance metrics, for high-speed implementations, is shown in Fig. 6 for Artix-7 and in Fig. 7 for Virtex-7. The exact results and relative differences among the candidates are also summarized in Tables 17 and 18. The primary metrics used for ranking are the execution times for

Table 17: Ranking of hardware implementations in terms of the execution time for encapsulation. For each algorithm, the first number represent the execution time in μs; the second number is the ratio of the execution time for a given algorithm and the best execution time in the given ranking.

Artix-7								
Level 1			**Level 3**			**Level 5**		
Round5_5d	12.2	1.00	**Kyber**	19.9	1.00	**Round5_5d**	27.6	1.00
Kyber	14.8	1.21	**LAC-v3b**	21.2	1.07	**LAC-v3b**	28.1	1.02
LAC-v3b	14.8	1.21	**Round5_5d**	21.6	1.09	**Kyber**	28.4	1.03
Round5_0d	16.0	1.31	Round5_0d	25.6	1.29	**NewHope**	30.3	1.10
NewHope	16.3	1.34	LAC-v3a	29.1	1.46	LAC-v3a	33.9	1.23
LAC-v3a	17.9	1.47						
Virtex-7								
Level 1			**Level 3**			**Level 5**		
LAC-v3b	9.6	1.00	**LAC-v3b**	14.1	1.00	**LAC-v3b**	22.1	1.00
Round5_5d	9.7	1.01	**Round5_5d**	16.9	1.20	**Round5_5d**	23.0	1.04
LAC-v3a	11.5	1.20	**Kyber**	17.1	1.21	**NewHope**	23.1	1.05
NewHope	12.4	1.29	LAC-v3a	20.1	1.43	**Kyber**	24.3	1.10
Round5_0d	12.5	1.30	Round5_0d	20.3	1.44	Round5_0d	26.9	1.22
Kyber	12.6	1.31				LAC-v3a	27.2	1.23

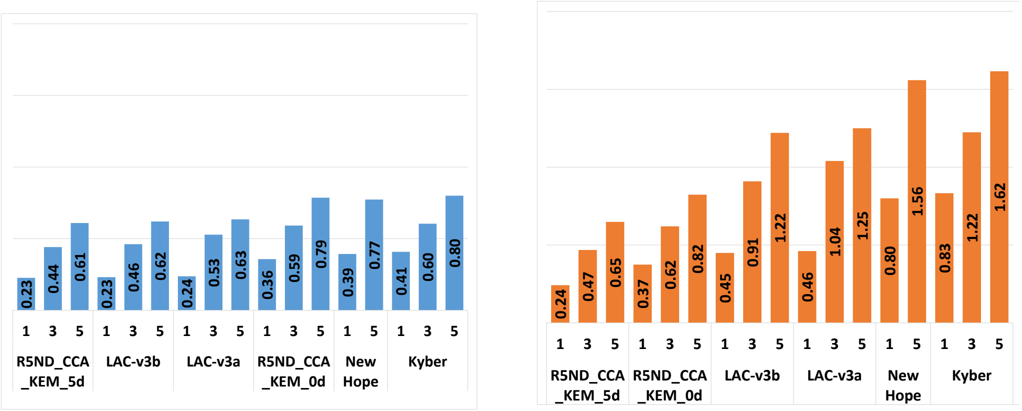
Table 18: Ranking of hardware implementations in terms of the execution time for decapsulation. For each algorithm, the first number represent the execution time in μs; the second number is the ratio of the execution time for a given algorithm and the best execution time in the given ranking.

Artix-7								
Level 1			**Level 3**			**Level 5**		
Round5_5d	16.3	1.00	**Kyber**	27.2	1.00	**Kyber**	36.2	1.00
LAC-v3b	18.9	1.16	**Round5_5d**	28.4	1.04	**Round5_5d**	36.4	1.01
Round5_0d	20.6	1.26	**LAC-v3b**	28.7	1.06	**LAC-v3b**	37.9	1.05
Kyber	21.4	1.31	Round5_0d	33.2	1.22	**NewHope**	41.5	1.15
NewHope	22.0	1.35	LAC-v3a	37.4	1.38	LAC-v3a	43.8	1.21
LAC-v3a	22.2	1.36						
Virtex-7								
Level 1			**Level 3**			**Level 5**		
LAC-v3b	12.4	1.00	**LAC-v3b**	19.1	1.00	**Kyber**	31.0	1.00
Round5_5d	13.3	1.07	**Round5_5d**	22.7	1.19	**Round5_5d**	31.2	1.01
LAC-v3a	14.4	1.16	**Kyber**	23.3	1.22	**NewHope**	31.7	1.02
Round5_0d	16.0	1.29	LAC-v3a	25.8	1.35	Round5_0d	35.8	1.15
NewHope	16.8	1.35	Round5_0d	27.0	1.41	**LAC-v3b**	37.7	1.22
Kyber	18.3	1.48				LAC-v3a	43.5	1.40



(a) (b)

Figure 4: (a) Public Key and (b) Private Key transfer latency (μs) on Artix-7



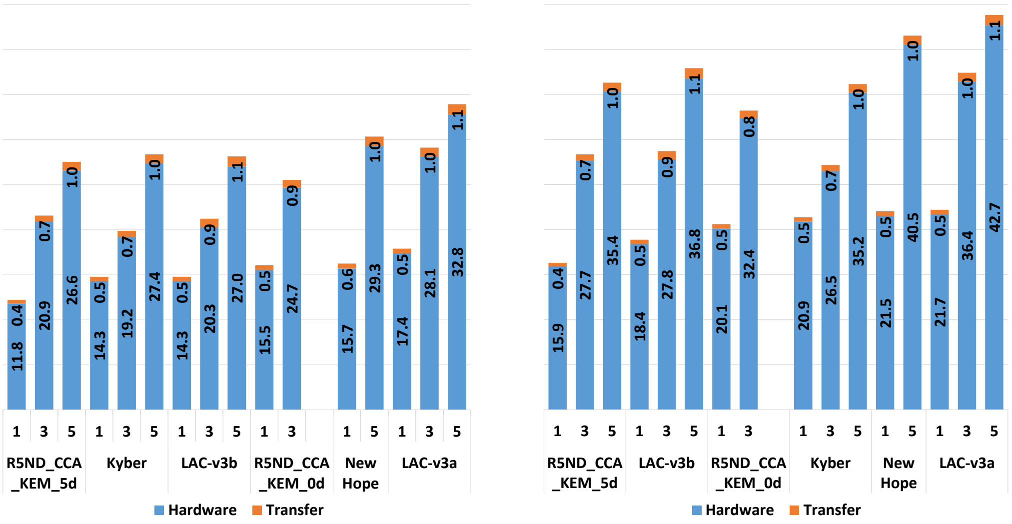
(a) (b)

Figure 5: (a) Public Key and (b) Private Key transfer latency (μs) on Virtex-7

encapsulation and decapsulation, respectively.

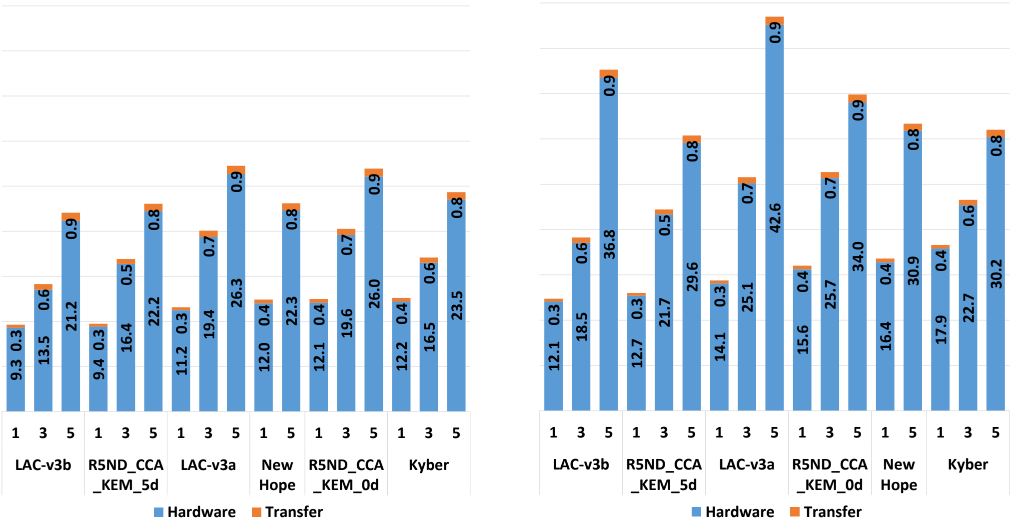
LAC-v3b, with q=256, clearly outperforms LAC-v3a, with q=251, in terms of both encapsulation and decapsulation time. Similarly, R5ND_CCA_KEM_5d (with error correction) outperforms R5ND_CCA_KEM_0d (without error correction). The relative ranking of Kyber, LAC-v3b, NewHope, and R5ND_CCA_KEM_5d, changes depending on the operation, security level, and FPGA family, but overall differences are minuscule. Thus, none of these four algorithms has a clear edge over the other in terms of hardware efficiency. Overall, the most efficient variants of all four candidates are in a virtual tie with one another.

In Table 19, we compare our hardware implementation of NewHope with the best high-speed implementation of this algorithm available to date. This implementation was described in [80], but it covered only a subset of the functionality of the IND-CCA KEM, namely the IND-CPA secure public-key encryption (PKE). Since for our own implementation, we could generate results for any subset of the complete CCA KEM design and using an arbitrary platform, the presented comparison is as fair as possible. Both sets of results concern exactly the same functionality, implemented using the same optimization target, with results generated using exactly the same platform.



(a) (b)

Figure 6: Execution Time for (a) Encapsulation and (b) Decapsulation (μs) on Artix-7



(a) (b)

Figure 7: Execution Time for (a) Encapsulation and (b) Decapsulation (μs) on Virtex7

Our implementation outperforms the design by Zhang et al. [80] in terms of all execution times. At security level 1, the speed-up varies between 2.2 for decryption, through 2.4 for key generation, to 2.6 for encryption. Similarly, at the security level 5, the speed-up varies between 2.0 for decryption, through 2.4 for key generation, to 2.6 for encryption. The penalty paid for this increase in speed is the increase in the number of LUTs by 33%, in flip-flops by a factor of 2.2, doubling the number of DSP units from 2 to 4, and increasing the number of BRAMs from 7-8 to 12. Overall, taking into the optimization for high-speed, our design is superior. However, the design by [80] provides an interesting example of trading the speed for a reduction in resource utilization.

Table 19: Comparison between this work and the best hardware design of NewHope reported in the literature to date. All results for Zynq-7000 SoC FPGA.

Design	Max. Freq.	Freq. Ratio	LUT	LUT Ratio	FF	FF Ratio	DSP	DSP Ratio	BR AM	BRAM Ratio	Key Generation				CPA Encryption				CPA Decryption			
											cycles	cycl. ratio	μs	μs ratio	cycles	cycl. ratio	μs	μs ratio	cycles	cycl. ratio	μs	μs ratio
NewHope-512 CPA-PKE																						
[80]	200	0.89	6,780	0.75	4,026	0.46	2	0.50	7	0.58	4,200	2.16	21.0	2.44	6,600	2.16	33.0	2.43	2,500	1.68	12.5	1.89
TW	225		9,009		8,768		4		12		1,946		8.6		3,061		13.6		1,487		6.6	
NewHope-1024 CPA-PKE																						
[80]	200	0.89	6,781	0.75	4,127	0.47	2	0.50	8	0.67	8,000	2.14	40.0	2.41	12,500	2.18	62.5	2.45	4,800	1.66	24.0	1.87
TW	225		9,009		8,768		4		12		3,738		16.6		5,746		25.5		2,892		12.9	

Table 20: Comparison of the execution times of major operations of the related CPA PKE and CCA KEM schemes when implemented in hardware using Artix-7. The ratio columns contain ratios of the execution times of Encapsulation/Encryption and Decapsulation/(Encryption+Decryption). All execution times are calculated without taking into account the time necessary to read inputs and offload outputs.

Algorithms	**CPA-PKE**				**CCA-KEM**					
	Encryption		**Decryption**		**Encapsulation**			**Decapsulation**		
	cycles	**us**	**cycles**	**us**	**cycles**	**us**	**ratio**	**cycles**	**us**	**ratio**
Kyber-512	2,602	12.4	1,608	7.7	2,995	14.3	1.15	4,395	20.9	1.04
Kyber-768	3,498	16.7	1,800	8.6	4,035	19.2	1.15	5,555	26.5	1.05
Kyber-1024	5,074	24.2	1,992	9.5	5,755	27.4	1.13	7,395	35.2	1.05
LAC-128-v3a	3,021	16.3	864	4.7	3,215	17.4	1.07	4,023	21.7	1.03
LAC-192-v3a	4,516	26.3	1,461	8.5	4,840	28.1	1.07	6,272	36.4	1.05
LAC-256-v3a	5,156	30.9	1,607	9.6	5,480	32.8	1.06	8,499	42.7	1.05
LAC-128-v3b	2,542	13.2	864	4.5	2,736	14.3	1.08	3,544	18.4	1.04
LAC-192-v3b	3,542	18.6	1,461	7.7	3,866	20.3	1.09	5,297	27.8	1.06
LAC-256-v3b	4,182	25.0	1,607	9.6	4,506	27.0	1.08	7,525	36.8	1.06
NewHope-512	3,061	13.6	1,487	6.6	3,538	15.7	1.16	4,829	21.5	1.06
NewHope-1024	5,746	25.5	2,892	12.9	6,583	29.3	1.15	9,111	40.5	1.05

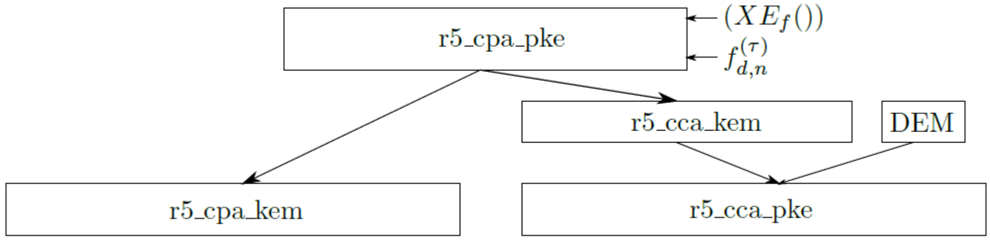


Figure 8: Dependencies between CPA and CCA versions of Round5 proposals [72]

Table 21: Comparison between CPA-KEM, CCA-KEM and CCA-PKE variants of R5ND_5d on Artix-7

Algorithm	**Max. Freq.**	**Encaps./Encrypt.**				**Decaps./Decrypt.**			
		cycles	**ratio**	**us**	**ratio**	**cycles**	**ratio**	**us**	**ratio**
CPA_1KEM	204	2,308	1.00	11.3	1.00	1,137	1.00	5.6	1.00
CCA_1KEM	204	2,492	1.08	12.2	1.08	3,328	2.93	16.3	2.93
CCA_1PKE	183	2,518	1.09	13.8	1.22	3,352	2.95	18.3	3.29
CPA_3KEM	169	3,582	1.00	21.2	1.00	1,726	1.00	10.2	1.00
CCA_3KEM	174	3,755	1.05	21.6	1.02	4,932	2.86	28.3	2.78
CCA_3PKE	163	3,782	1.06	23.2	1.09	4,956	2.87	30.4	2.98
CPA_5KEM	154	4,435	1.00	28.8	1.00	2,123	1.00	13.8	1.00
CCA_5KEM	169	4,655	1.05	27.5	0.96	6,137	2.89	36.3	2.63
CCA_5PKE	156	4,683	1.06	30.0	1.04	6,161	2.90	39.5	2.86

In Table 20, we compare the execution times of the CCA-KEM schemes with the execution times of the underlying CPA-PKE schemes, for Kyber, LAC, and NewHope. The ratios of the encapsulation and encryption times vary between to 1.07 and 1.17. That means that the PKE encryption is a dominant operation, and an overhead of other operations does not exceed 17%. For all four KEMs listed in this table, decapsulation includes one call to decryption and one call to encryption. Thus, the ratio listed under Decapsulation is a ratio of the execution time of decapsulation over the sum of the execution times of encryption and decryption. This ratio varies between 1.03 and 1.06, which means that the overhead of remaining operations does not exceed 6%.

In Table 21, we compare our hardware implementations of different schemes of the Round5 proposal. The dependencies among these schemes are graphically illustrated in Fig. 8. Our comparison contains results for maximum clock frequency and the execution times of major operations in the CPA-KEM, CCA-KEM, and CCA-PKE schemes. These metrics illustrate the cost of additional security of CCA-KEM as compared to CPA-KEM. The biggest difference is in the execution time of the CCA-KEM decapsulation and the CCA-PKE decryption, as compared to the CPA-KEM decapsulation. This difference comes from the Fujisaki-Okamoto transformation, used for providing CCA security. In CCA-KEM, during decapsulation, one additional CPA-PKE encryption is performed. CCA-KEM and CCA-PKE use the same parameter set, but CCA-PKE includes CCA-KEM and performs additional symmetric-key encryption after encapsulation of the key used for it. The differences in the execution times of the CPA-KEM and CCA-KEM encapsulations and the CCA-PKE encryption are negligible.

5.2 Profiling the best available software implementations in C

Table 22: Source of software implementations

Algorithm	**Software Source**	**Ref**	**Opt**
FrodoKEM	https://github.com/Microsoft/PQCrypto-LWEKE	✓	✓
Kyber	https://github.com/pq-crystals/kyber	✓	
LAC	https://github.com/pqc-lac/lac-intel64	✓	✓
NewHope	https://github.com/newhopecrypto/newhope	✓	
NTRU	https://github.com/jschanck/ntru	✓	
NTRU Prime	https://bench.cr.yp.to/supercop.html	✓	✓
Round5	https://github.com/r5embed/r5embed	✓	✓
Saber	https://github.com/KULeuven-COSIC/SABER	✓	

We implemented 12 CCA-secure KEMs representing eight Round 2 lattice-based candidates using the software/hardware co-design approach described in detail in Section 4. In Table 22, we list the repositories containing C source code used as a starting point for our software/hardware implementations. In the case of four candidates - FrodoKEM, LAC, NTRU Prime, and Round5 - optimized implementations in C, different than reference implementations exist. For the remaining four candidates, their best portable implementations are the same (or almost the same) as their reference implementations submitted at the beginning of Round 2. We used the mentioned above implementations in C as a starting point for our first software implementation of each of the 12 implemented KEMs, ported to ARM Cortex-A53 using the procedure described in Section 4.5.

The results of profiling for the obtained pure-software implementations, running on a single core of ARM Cortex-A53, at the frequency of 1.2 GHz, are presented in the left portions of Tables 29, 30, 31, 32, 33, 34, 35, 36, 37, and 38, in Appendix A.

For each of the 12 investigated algorithms and each major operation (Encapsulation and Decapsulation), two to five most time-consuming functions are identified. For each of these functions, we provide their execution time (in microseconds) and the percentage of the total execution time. In the right portions of the same tables, we list in bold functions offloaded to hardware. For the functions combined together, they are listed in the same field of the table, with sub-indices, such as 1.1, 1.2, 1.3, etc. A single execution time and a single percentage of the software/hardware execution time is given for such a combined function.

It is important to note that the execution time of all functions offloaded to hardware, listed in Tables 29–38 include both the execution time in hardware as well as the time necessary to transfer control, inputs, and outputs between the processor and a hardware accelerator. It should also be mentioned that the number of functions offloaded to hardware may be misleading, as these functions may appear at different levels of hierarchy. For example, for the encapsulation in Kyber, only two functions are offloaded. However, these are function involving the majority of operations of Kyber, amounting to 99.55-99.81% of the total execution time in the software-only implementation. For all algorithms, at least the first and the second most time-consuming functions are offloaded to hardware.

The total percentage of the execution time taken by a portable software implementation to execute operations offloaded to hardware is shown in Figs. 9 and 10.

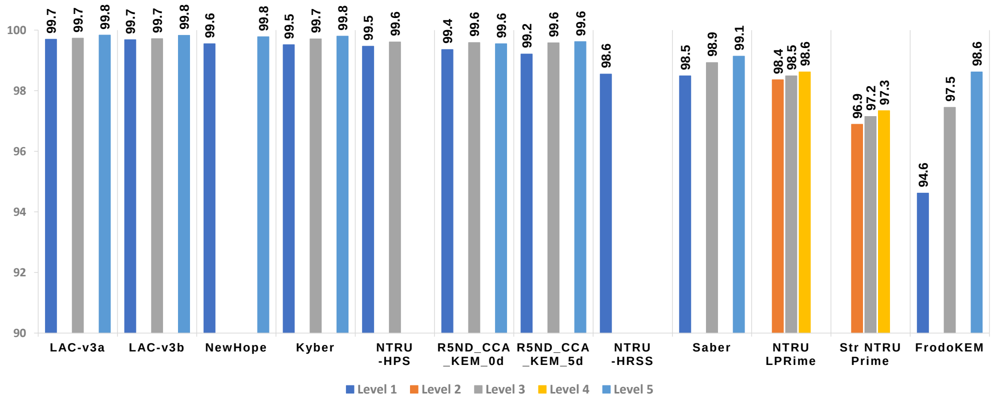


Figure 9: Encapsulation: The Software Part Sped Up by Hardware [%]

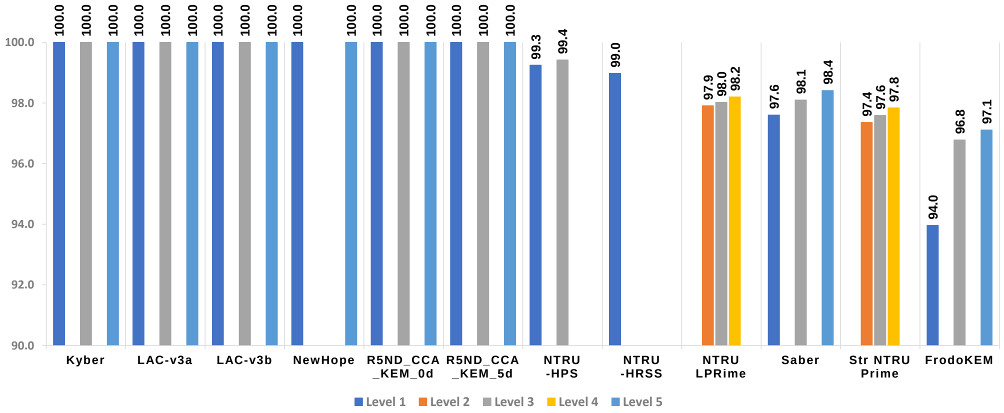


Figure 10: Decapsulation: The Software Part Sped Up by Hardware [%]

5.3 Results for Software/Hardware Implementations

Twelve hardware accelerators developed using the methodology described in Section 4 are characterized in Table 23 using their maximum clock frequency and resource utilization

Table 23: Maximum frequency and resource utilization of hardware accelerators developed as a part of software/hardware co-designs targeting Zynq Ultrascale+

Algorithm	Security Category: Parameter Set	Max. Freq.	LUT	FF	Slice	DSP	BR AM
FrodoKEM	Frodo-640	402	7,213	6,647	1,186	32	13.5
FrodoKEM	Frodo-976	402	7,087	6,693	1,190	32	17.0
FrodoKEM	Frodo-1344	417	7,015	6,610	1,215	32	17.5
Kyber	KYBER512	410	12,034	10,532	2,327	8	14.0
Kyber	KYBER768	405	12,195	10,461	2,253	8	14.0
Kyber	KYBER1024	405	12,589	12,574	2,635	8	14.0
LAC-v3a	LAC-128	385	25,123	16,005	3,720	0	8.5
LAC-v3a	LAC-192	370	41,898	26,233	6,134	0	11.5
LAC-v3a	LAC-256	357	46,756	26,989	6,774	0	11.5
LAC-v3b	LAC-128	400	18,311	15,966	2,672	0	8.5
LAC-v3b	LAC-192	385	27,209	26,193	4,024	0	11.5
LAC-v3b	LAC-256	357	33,234	26,567	4,889	0	11.5
NewHope	NEWHOPE512-CCA-KEM	490	9,307	8,928	1,721	4	11.0
NewHope	NEWHOPE1024-CCA-KEM	490	9,307	8,928	1,721	4	11.0
NTRU-HPS	ntruhps2048677	200	42,578	22,717	8,235	677	8.5
NTRU-HPS	ntruhps4096821	200	49,735	30,599	9,924	821	8.5
NTRU-HRSS	ntruhrss701	200	48,773	25,178	8,110	701	2.5
NTRU LPRime	kem/ntrulpr653	278	45,901	39,426	8,938	0	8.0
NTRU LPRime	kem/ntrulpr761	263	55,054	45,133	9,769	0	8.0
NTRU LPRime	kem/ntrulpr857	250	64,022	50,120	10,554	0	8.0
Str NTRU Prime	kem/sntrup653	278	62,797	33,531	9,110	0	9.0
Str NTRU Prime	kem/sntrup761	263	70,066	38,144	10,319	0	9.0
Str NTRU Prime	kem/sntrup857	250	78,379	42,274	11,509	0	9.0
Round5	R5ND_CCA_1KEM_0d	294	52,589	80,875	10,154	0	3.0
Round5	R5ND_CCA_3KEM_0d	267	72,870	107,748	13,360	0	3.0
Round5	R5ND_CCA_5KEM_0d	250	99,310	156,732	18,095	0	3.0
Round5	R5ND_CCA_1KEM_5d	357	38,116	56,189	7,538	0	3.0
Round5	R5ND_CCA_3KEM_5d	294	54,532	95,436	12,395	0	3.0
Round5	R5ND_CCA_5KEM_5d	238	69,254	114,007	12,774	0	3.0
Saber	LightSaber-KEM	322	12,343	11,288	1,989	256	3.5
Saber	Saber-KEM	322	12,566	11,619	1,993	256	3.5
Saber	FireSaber-KEM	322	12,555	11,881	2,341	256	3.5

when implemented on Xilinx Zynq UltraScale+ SoC FPGA. All results have been obtained after placing and routing.

NewHope, Kyber, and FrodoKEM are able to achieve the highest clock frequencies, above 400 MHz for all parameter sets. LAC has frequencies between 350 and 400 MHz, depending on a variant and security level. The maximum frequency of Round5 decreases significantly with the increase in the security level, especially for a version with the error-correcting code, where the frequency drops from 357 MHz for security level 1 to 238 MHz for security level 5. On the other hand, Saber has the same clock frequency, 322 MHz, for all of its parameter sets. The operating frequencies for the two variants of NTRU Prime are in the range 250-280. They are limited mainly by the reduction modulo q. To reduce numbers with the prime modulus q, we selected the conditional subtraction method, which is relatively simple but comes with a long critical path. NTRU-HPS and NTRU-HRSS have the lowest clock frequency of 200 MHz. These frequencies are affected by the logic for converting polynomials from R/q to S/q and from R/q to $S/3$.

The accelerators for NTRU-HPS and NTRU-HRSS involve the highest number of integer multiplications performed in parallel. These multiplications in the FPGA fabric are delegated to dedicated DSP units. The DSP units are also taken advantage of in Saber and to a lower extent in FrodoKEM, Kyber, and NewHope. LAC, Round5, NTRU LPRime, and Streamlined NTRU Prime do not involve any integer multiplications in hardware. This is because the coefficients of one of the multiplied polynomials always belong to the set {-1, 0, 1}.

FrodoKEM is the algorithm with the highest utilization of BRAMs, which reaches 17.5 blocks. The algorithms with the lowest utilization of BRAMs (between 2.5 and 3.5) include NTRU-HRSS, Round5, and Saber. The remaining KEMs require 8–14 BRAMs.

Round5, Streamlined NTRU Prime, and NTRU LPRime use the largest number of LUT, flip-flops (FFs), and Slices. FrodoKEM, NewHope, Kyber, and Saber use the smallest number. The amount of resources used increases noticeably with the increase in the security level for 8 out of 12 KEMs. The following algorithms have a desirable property that the security levels do not substantially affect resource utilization (except for the small increase in the number of BRAMs in FrodoKEM): FrodoKEM, Kyber, NewHope, and Saber.

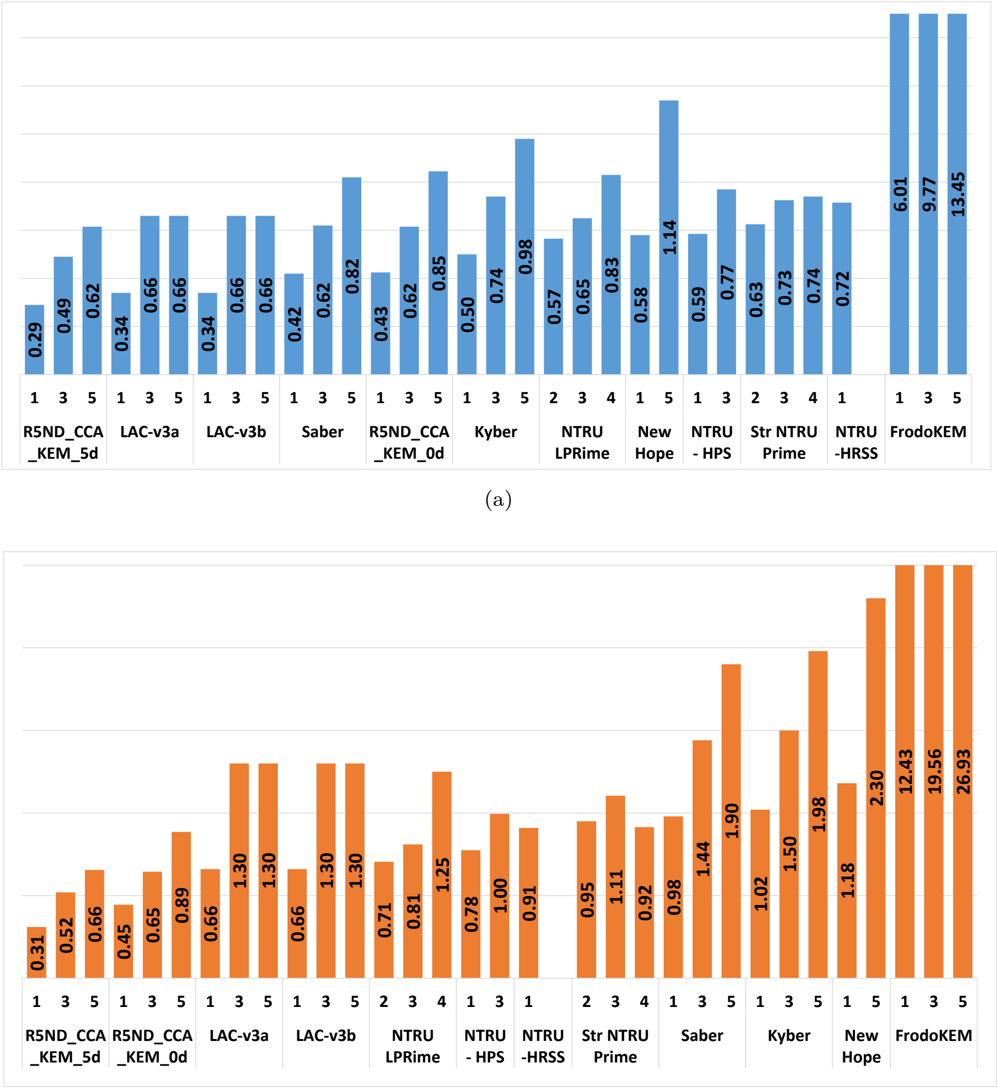
Because of the timing dependencies, and in particular, the bottleneck caused by SHAKE, our implementation of FrodoKEM cannot be easily sped up by trading additional resources for speed. This example clearly illustrates the potential algorithmic limits on the amount of parallelization (and thus the maximum speed-up), which is independent of the amount of hardware resources available to the designer.

The times necessary to load a public key (required for encapsulation) and a secret (private) key (required for decapsulation) are proportional to the size of the respective key and inversely proportional to the maximum clock frequency of a given PQC unit. All transfers are assumed to be conducted using a 64-bit infifo_data bus. The sizes of keys for all variants of all investigated algorithms are summarized in Table 14. The maximum clock frequencies are listed in Table 23. In Fig. 11, we compare these key loading times for all 12 implemented KEMs. R5ND_CCA_KEM_5d has the shortest key-loading times, and FrodoKEM the longest. However, the differences among these times are relatively minor for all KEMs other than FrodoKEM. They do not exceed a factor of 2 for loading a public key, and 4 for loading a private key. Except for NewHope at the security level 5 and FrodoKEM at all security levels, the public-key loading times stay below 1 μs, and private-key loading times below 2 μs.

Total execution times of our software/hardware implementations are summarized in Fig. 12 for encapsulation, and Fig. 13 for decapsulation.

Rankings can be considered separately for three groups of parameter sets listed in Table 14, with the security levels 1 and 2, 3 only, and 4 and 5, respectively. Only the first group contains all 12 investigated algorithms. In the second group, NTRU-HRSS and NewHope are missing, and in the third group, NTRU-HRSS and NTRU-HPS are not represented. In Figs. 12 and 13, KEMs are arranged according to their ranking for security levels 1 and 2.

Each execution time is separated into three components: the execution time in hardware (i.e., in the hardware accelerator located in programmable logic of Zynq UltarScale+ SoC FPGAs), the time required to transfer data and control between the processor and the hardware accelerator, and the execution time in software (i.e., in ARM Cortex-A53). For encapsulation, at least the function `randombytes()` is assumed to be executed in software to generate a seed for a deterministic random bit generator (typically based on SHAKE) implemented in hardware. For decapsulation, no internal function of KEM has to be executed in software. We treat implementation as a software/hardware implementation even if the operation of the processor is limited only to sending KEM inputs to and receiving KEM outputs from the hardware accelerator. Hence, our software/hardware implementations of LAC-v3b, R5ND_CCA_KEM_5d, NewHope, Kyber, LAC-v3a, and R5ND_CCA_KEM_0d, which have the shortest execution times of encapsulation and decapsulation are based on the pure hardware implementations of KEMs, described in Section 5.1. The ranking of these six KEMs is similar, but not identical to the ranking of their corresponding hardware implementations. The small changes in rankings come



(a)

(b)

Figure 11: (a) Public Key and (b) Private Key transfer latency (μs) of SW/HW co-design on Zynq-Ultrascale+

from small differences in the transfer time and execution time in software, as well as from the different maximum clock frequency of hardware accelerators when implemented in programmable logic of Zynq UltraScale+ rather than Artix-7 or Virtex-7 (as in Figs. 6 and 7). Overall, however, not leaving any operation (other than `randombytes()`) in software gives these 6 KEMs enough advantage to outperform all six remaining schemes.

FrodoKEM is by far the slowest KEM, and it cannot outperform any other scheme even if 100% of its operations are moved to hardware. For encapsulation, NTRU-HPS, Streamlined NTRU Prime, and NTRU LPRime are also very unlikely to move in ranking ahead of any of the first six schemes, because even after reducing their execution time in software to zero and making the transfer time similar to the transfer time of the first six schemes (i.e., in the range of 6.2-7.0 mus), their execution times would exceed the overall time for the KEM at position 6, R5ND_CCA_KEM_0d.

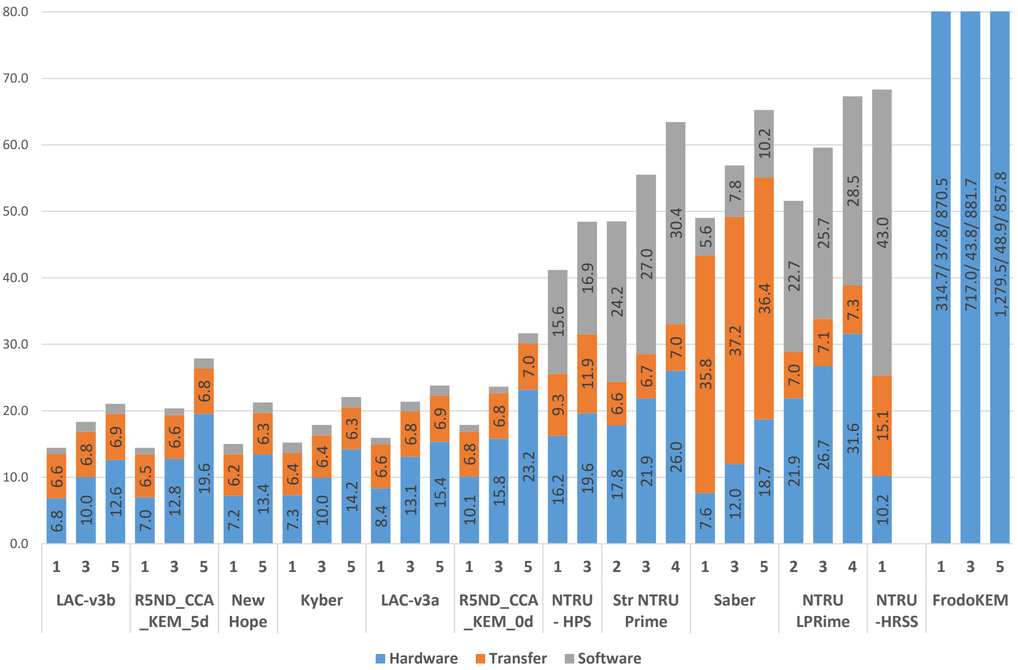


Figure 12: Encapsulation: Total Execution Time in Software/Hardware [μs]

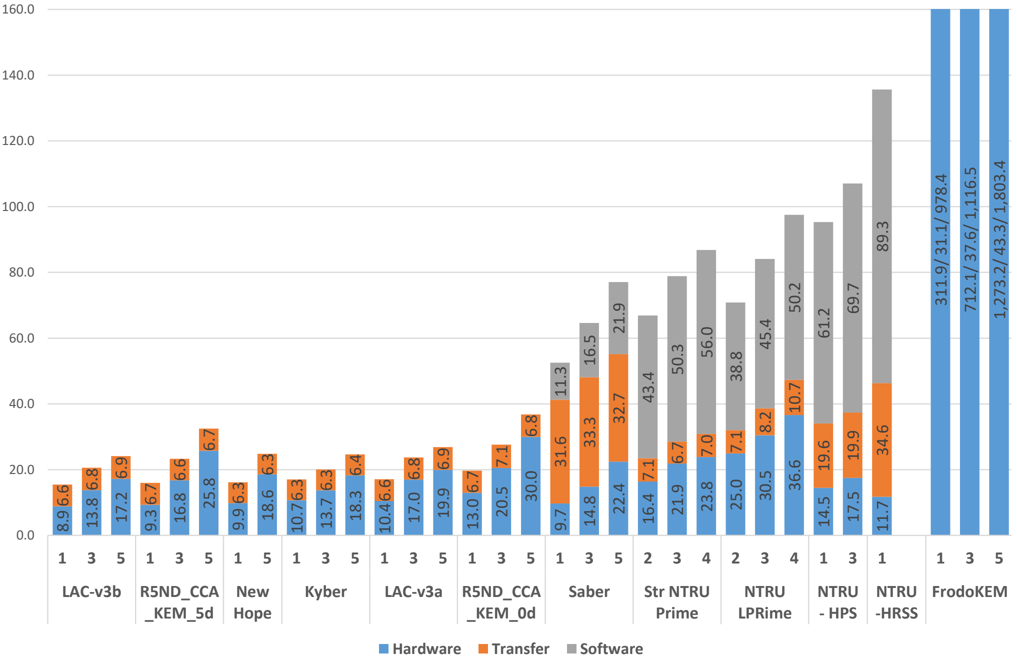


Figure 13: Decapsulation: Total Execution Time in Software/Hardware [μs]

For Saber and NTRU-HRSS, it is too early to make such a judgment. However, the presented results at least reveal some potential weaknesses of these two algorithms (from the point of view of ease of their software/hardware partitioning), which can be observed by analyzing their profiling results, summarized in Tables 38 and 34. For NTRU-HRSS, even after moving to hardware its four most time-consuming operations, the software still amounts to a significant percentage of the total execution time. In Saber, even after moving to hardware its five most time-consuming operations, transfer time still dominates the total execution time. This last point can be reinforced by analyzing Table 24. According to this table, our software/hardware implementation of Saber has the largest number of transfers between the processor and the accelerator (6). Disregarding FrodoKEM, which is very slow in hardware already, NTRU-HRSS is the only other algorithm that requires more than one transfer during the enacapsulation.

Table 24: Data Transfer Summary for SW-HW co-designs

Algorithms	**Encapsulation**			**Decapsulation**		
	Count	**Load (bytes)**	**Return (bytes)**	**Count**	**Load (bytes)**	**Return (bytes)**
KYBER512	1	32	768	1	736	32
KYBER768	1	32	1,120	1	1,088	32
KYBER1024	1	32	1,600	1	1,568	32
R5ND_1KEM_0d	1	16	740	1	724	16
R5ND_3KEM_0d	1	24	1,103	1	1,079	24
R5ND_5KEM_0d	1	32	1,509	1	1,477	32
R5ND_1KEM_5d	1	16	620	1	604	16
R5ND_3KEM_5d	1	24	934	1	910	24
R5ND_5KEM_5d	1	32	1,285	1	1,253	32
LightSaber-KEM	6	1,600	704	5	1,216	896
Saber-KEM	6	2,272	960	5	1,216	1,280
FireSaber-KEM	6	2,976	1,216	5	1,600	1,664
Frodo-640	4	19,400	22,032	3	9,768	22,008
Frodo-976	4	31,472	31,328	3	15,816	31,304
Frodo-1344	4	42,780	43,136	3	21,728	43,104
LAC-128-v3a	1	16	736	1	704	32
LAC-192-v3a	1	32	1,384	1	1,352	32
LAC-256-v3a	1	32	1,496	1	1,464	32
LAC-128-v3b	1	16	736	1	704	32
LAC-192-v3b	1	32	1,384	1	1,352	32
LAC-256-v3b	1	32	1,496	1	1,464	32
NEWHOPE512	1	32	960	1	928	32
NEWHOPE1024	1	32	1,856	1	1,824	32
ntruhps2048677	1	720	1,372	2	2,318	1,408
ntruhps4096821	1	864	1,708	2	2,906	1,692
ntruhrss701	2	1,024	1,436	3	2,854	1,512
kem/sntrup653	1	40	1,448	1	912	72
kem/sntrup761	1	40	1,664	1	1,048	72
kem/sntrup857	1	40	1,856	1	1,192	72
kem/ntrulpr653	1	40	1,584	1	1,040	72
kem/ntrulpr761	1	40	1,800	1	1,176	72
kem/ntrulpr857	1	40	1,992	1	1,320	72

For decapsulation, the execution time in software is eliminated entirely for the first six KEMs in the ranking. The transfer time is similar for all algorithms from that group. The transfer time dominates the execution time in Saber, because, as shown in Table 24, five transfers are required, more than for any other algorithm. The transfer time is also unusually long for NTRU-HRSS and NTRU-HPS (with 3 and 2 transfers, respectively). As a result, it might be too early to judge whether Saber, NTRU-HRSS, and NTRU-HPS can be made as efficient as the first six KEMs, after moving all their operations to hardware. On the other hand, for NTRU LPRime and Streamlined NTRU Prime there is already a strong indication that these algorithms will not be able to move ahead of any of the first six KEMs, even if implemented entirely in hardware. Finally, FrodoKEM is by far the slowest algorithm out of all 12 implemented in this study.

5.4 Use of High-Level Synthesis

A traditional approach to high-level synthesis is based on starting from the existing implementation in C, C++, or System C, and then introducing modifications aimed at:

- inferring the desired interface
- optimizing speed
- minimizing resource utilization.

In the case of PQC candidates, a starting point is naturally determined by either reference implementation or the best portable implementation written entirely in C, such as those described in Section 4.5, used as a starting point for the software/hardware co-design.

Traditionally, a significant percentage of all modifications amounts to guiding the synthesis tool toward the desired outcome using the C language pragmas, ignored by traditional high-level language compilers, but treated as directives by a given synthesis tool. This approach is used, in particular, in two popular HLS tools targeting FPGAs, Vivado HLS and LegUp. For example, there are over 20 *pragma* directives in the current version of Vivado HLS. Their different combinations lead to different hardware architectures. The impact of a particular *pragma* directive is heavily dependent on the code structure and the algorithm. Some directives may have no impact at all; others may dramatically change the speed vs. cost trade-off. Exploring all possible combinations is often unrealistic. Additionally, in many cases, code refactoring may give better results than an optimal choice and placement of directives.

The first attempt at applying HLS to benchmarking PQC schemes was reported in [14]. Only a few directives aimed at accomplishing unrolling of loops and pipelining were applied. The authors also attempted to synthesize the C code of the entire algorithm. Taking into account the limited availability of RTL code, no comparison with the equivalent RTL code was attempted.

An outcome of this approach is quite clearly illustrated in Table 25. The differences in obtained results are huge, although probably not that surprising, taking into account the almost complete reliance on tools in [14]. The HLS-based designs are bigger than RTL-based designs in terms of the number of LUTs by a factor of at least 144 for Kyber, 14.5 for NewHope, and 7.5 for Classic McEliece. These factors are obtained by dividing the area of the decapsulation/decryption unit in the HLS-based approach by the area of the combined unit, capable of performing key generation, encapsulation/encryption and decapsulation/decryption in the RTL approach. Thus, if units with the same functionality were compared, the ratios could be even higher. The HLS to RTL ratios of the encapsulation/encryption times are 10.5 for Kyber, 93 for NewHope, and 700 for Classic McEliece. For decapsulation/(decryption+encryption), the corresponding ratios are 36 for Kyber, 741 for NewHope, and 551 for Classic McEliece. Overall in terms of the latency times area product, the HLS-based designs are three orders of magnitude worse. Additionally, a significant difference in specific ratios for Kyber and NewHope, combined with the almost identical performance and resource usage of RTL designs, indicates that the approach pursued in [14] cannot correctly predict the relative ranking of PQC candidates unless the differences among them are truly enormous.

Table 25: Comparison between HLS-based designs from [14] and the corresponding RTL-based designs. All results for Virtex-7 FPGAs. TW denotes this work.

Design	Max. Freq.	Freq. Ratio	LUT	LUT Ratio	FF	FF Ratio	DSP	BRAM	Key Generation		Encaps./Enc.				Decaps./(Dec.+Enc.)^{cpa}			
									cycles	μs	cycles	cycl. ratio	μs	μs ratio	cycles	cycl. ratio	μs	μs ratio
Kyber-512 CCA-KEM																		
[14]	67	0.27	1,307,815 1,977,896	95.15 143.90	11,699 194,126	1.05 17.48	-	-	-	-	31,669	10.57	475.0	38.86	43,018	9.79	645.3	35.97
TW	245		13745		11107		8	14	2,160	8.8	2,995		12.2		4,395		17.9	
NewHope-512 CCA-KEM																		
[14]	67	0.23	136,457 164,937	12.03 14.54	25,639 28,999	2.90 3.28	0	0	-	-	307,847	87.01	4,617.7	385.03	721,986	149.51	10,829.8	661.58
TW	295		11,345		8,838		4	10.0	2,122	7.2	3,538		12.0		4,829		16.4	
mceliece6960119 CPA-PKE																		
[14]	100	0.77	840,430 870,908	7.19 7.45	60,270 79,962	0.32 0.42	-	-	-	-	3,787,729	699.75	37,877.3	908.33	10,659,024	424.07	106,590.2	550.85
[74]	130		116,928		188,324		0	607.0	974,306	7,500.4	5,413		41.7		25,135		193.5	

Table 26: Comparison between HLS and RTL method for NTT implementations.

Method	DSP	BRAM	LUT	FF	Slice	Max. Freq.	Latency (cycles)
NewHope-512 with 1 NTT module							
HLS	4	3	1,181	1,403	239	454	3,247
RTL	4	3	1,040	940	190	476	3,247
HLS/RTL	1.00	1.00	1.14	1.49	1.26	0.95	1.00
NewHope-1024 with 1 NTT module							
HLS	4	5	1,110	1,342	219	455	6,266
RTL	4	5	842	803	170	476	6,266
HLS/RTL	1.00	1.00	1.32	1.67	1.29	0.96	1.00
Kyber512 with 2 NTT modules							
HLS	24	7	2,325	2,346	430	455	1,271
RTL	24	5	2,040	3,223	433	500	1,271
HLS/RTL	1.00	1.40	1.14	0.73	0.99	0.91	1.00
Kyber768 with 3 NTT modules							
HLS	36	11	5,379	4,043	1,074	416	1,271
RTL	36	7.5	3,054	5,098	637	500	1,271
HLS/RTL	1.00	1.47	1.76	0.79	1.69	0.83	1.00
Kyber1024 with 4 NTT modules							
HLS	48	14	7,111	5,457	1,374	416	1,271
RTL	48	10	4,055	6,803	960	500	1,271
HLS/RTL	1.00	1.40	1.75	0.80	1.43	0.83	1.00

As a result, a substantially different approach was needed to overcome this inefficiency. This approach was demonstrated by our group in [22], [24], [56], and [57]. First, HLS is combined with software/hardware co-design. This way, only the most time-consuming operations (and preferably a single operation) needs to be offloaded to hardware. These operations can be identified using techniques described in Section 4.6. Secondly, these critical operations are described using block diagrams. Third, the block diagrams are translated into HLS-ready C code, written from scratch, and enhanced with HLS directives encoded using pragmas. The designer then debugs the code using a C testbench, which is much easier to develop and easier to use than HDL testbench. When the code is determined to be functionally correct, it is passed through synthesis. If the number of clock cycles is different from the expected number obtained from the analysis of the block diagram, additional pragmas need to be added, or the code needs to be refactored (rewritten) to make it more suitable for HLS tools. For example, a programmer may apply explicit function sharing or eliminate dependencies preventing multiple operations from executing in parallel. These optimizations continue at least until the required number of clock cycles is reached. They may also be applied to reduce the number of specific logic resources, such as LUTs, DSP units, or BRAMs, as well as to increase the maximum clock frequency.

Below, we demonstrate the application of this approach to the preliminary software/hardware implementations of two Round 2 candidates: NewHope and Kyber. In both implementations, we decided to offload to hardware only the most time-consuming operation, the Number Theoretic Transform (NTT). This operation was first expressed using a detailed block diagram, presented in [57]. Then, the described above methodology was followed. In parallel, optimized RTL implementation was developed for the purpose of evaluating the quality of our HLS design. The obtained results are summarized in Table 26.

These results indicate that our primary goal of reaching the same number of clock cycles as that obtained using the RTL approach was accomplished. At the same time, the clock frequency was lower by up to 17%, and the number of LUTs, flip-flops, and slices higher by up to 76%, 67%, and 69%, respectively.

Overall, the RTL- and HLS-based approaches to the design of a hardware accelerator for NTT led to almost the same total speed-up of the software/hardware implementation. At the same time, the development time was several times shorter for the HLS-based approach.

The disadvantage of our approach is the need for a detailed block diagram, which requires either hardware expertise within a team of HLS programmers or collaboration with a group of hardware designers. Additionally, most of the HLS-ready code needs to written from scratch. The reduction in clock frequency plays a secondary role, as it typically does not significantly affect the overall speed-up of the software/hardware implementation over the portable C code. Similarly, for high-speed implementations, the exact resource utilization plays a secondary role and does not affect the ranking of candidates.

5.5 Results for Software Implementations Optimized Using NEON Instructions of ARM

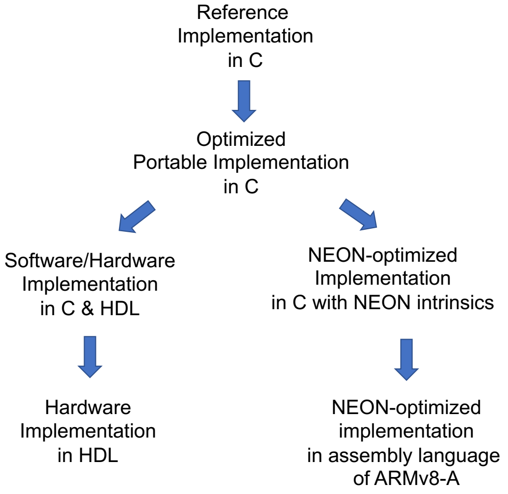


Figure 14: Types of optimized implementations.

Table 27: Comparison between Software Implementations using NEON instructions and Software-Hardware co-designs.

Algorithm	SW-Ref Encaps.	SW-Ref Decaps.	SW-Neon Encaps.	SW-Neon Decaps.	SW-HW Encaps.	SW-HW Decaps.	Ref/Neon Encaps.	Ref/Neon Decaps.	Ref/SW-HW Encaps.	Ref/SW-HW Decaps.	Neon/SW-HW Encaps.	Neon/SW-HW Decaps.
	μs	μs	μs	μs	μs	μs	Ratio	Ratio	Ratio	Ratio	Ratio	Ratio
NewHope-1024	723.5	891.2	338.3	363.6	21.3	24.9	2.1	2.5	34.0	35.8	**15.92**	**14.61**
ntruhrss701	2,964.5	8,789.8	123.7	252.1	68.3	135.6	24.0	34.9	43.4	64.8	**1.81**	**1.86**
ntruhps2048677	2,961.5	8,174.9	337.7	339.5	41.2	95.3	8.8	24.1	71.9	85.8	**8.20**	**3.56**
ntruhps4096821	4,285.1	11,981.7	401.4	434.3	48.4	107.1	10.7	27.6	88.5	111.9	**8.29**	**4.06**
LightSaber	522.4	655.2	153.2	166.6	49.0	52.5	3.4	3.9	10.6	12.5	**3.12**	**3.17**
Saber	1,008.3	1,210.8	268.8	291.7	56.9	64.7	3.8	4.2	17.7	18.7	**4.72**	**4.50**
FireSaber	1,643.3	1,919.9	420.8	456.8	65.2	77.1	3.9	4.2	18.1	17.9	**6.45**	**5.92**

On the selected platform, Zynq UltraScale+, a reference implementation of a PQC scheme in C can be optimized using several approaches shown in Fig. 14. First, basic optimizations may be still possible in C without affecting the portability of the code. From here, two divergent paths are worth investigating. First, Optimized Portable Implementation in C can be turned into a Software/Hardware Implementation in C and HDL, using the methodology described in this paper in Sections 4.3, 4.4, 4.6, 4.7. After all C functions are moved to hardware, this implementation becomes a pure hardware implementation in HDL.

An alternative path is based on the use of SIMD instructions of ARMv8-a, referred to as NEON instructions. These instructions can be called from C using the so-called intrinsics. NEON intrinsics are function calls that the compiler replaces with an appropriate NEON instruction or a sequence of NEON instructions. Intrinsics provide almost as much control as writing assembly language but leave the allocation of registers to the compiler [6]. Operations that cannot take advantage of vector instructions are left in C. This path can be further extended into pure assembly language code. This code may consist of hand-coded NEON assembly language instructions, as well as remaining (so-called scalar) assembly language instructions of ARMv8-a.

We have developed NEON-optimized implementations in C with NEON intrinsics for 4 investigated KEMs, representing 3 Round 2 PQC candidates, namely NewHope, NTRU-HPS, NTRU-HRSS, and Saber. Our starting point consisted of optimized implementations of these algorithms, targeting Intel and AMD processors, using AVX2 (Advanced Vector Extensions 2). In Table 27, we compare the performance of NEON-optimized software implementations (based on intrinsics) with the performance of our software/hardware implementations.

Our software/hardware implementations appear to be superior for all investigated candidates and parameter sets. Compared to the software implementation based on NEON intrinsics, the execution times for NewHope-1024 are over 14.5 times smaller in software/hardware. For NTRU-HPS software/hardware implementation is about 8 times faster for encapsulation and 3.5-4.0 times faster for decapsulation. Saber has the ratios approximately the same for encapsulation and decapsulation. However, the advantage of software/hardware increases with the increase in the security level. Finally, NTRU-HRSS has the smallest ratios in the range of 1.8-1.9.

Multiple conference papers have been devoted to the NEON-based implementation of a single public-key cryptosystem [16, 64, 65, 10, 51, 68, 66]. These papers demonstrate that developing optimized implementations based on NEON intrinsics, hand-coded NEON assembly language code, and hand-coded ARMv8-A RISC assembly language code is at least as complex and labor intensive as the development of optimized software/hardware implementations.

The advantages of NEON-based implementations include a) software-only paradigm - no need for expertise in hardware and knowledge of HDLs, b) NEON vector instructions run at a higher clock frequency than an FPGA-based hardware accelerator, c) using the NEON co-processor involves minimal (if any) transfer overhead, d) the NEON co-processor can be potentially reused for non-cryptographic operations, such as signal and image processing.

The primary advantages of the software/hardware implementations are: a) Programmable logic is much more powerful and less restrictive than the NEON co-processor in terms of the number and type of operations that can be executed in parallel. As a result, a higher overall speed-up is accomplished. b) Hardware written in HDL is likely to be more portable than software written in the assembly language of a particular processor. In particular, our software/hardware implementations can be ported to any other modern SoC FPGA, assuming that the amount of the required hardware resources does not exceed the capabilities of the programmable logic of a given SoC device.

6 Comparison with performance of the AVX2-optimized software implementations

In Table 28, we compare the performance of our software/hardware implementations, running on Zynq UltraScale+, with the performance of the best software implementations available to date, running on Intel Xeon E3-1220 v3 (3.1 GHz).

When comparing these implementations, one needs to keep in mind that the software

Table 28: Comparison of the GMU software/hardware implementations, running on Zynq UltraScale+, with the software implementations in supercop-20200525 running on Intel Xeon E3-1220 v3 (3100MHz)

Algorithm	median cycles	SW (us)	SW/HW (us)	Ratio
Encapsulation				
Level 1 & 2				
ntruhrss701	26116	8.4	68.3	0.12
ntruhps2048677	35352	11.4	41.2	0.28
kyber512	44404	14.3	15.2	0.94
sntrup653	46620	15.0	48.5	0.31
lightsaber2	67568	21.8	49.0	0.44
ntrulpr653	69400	22.4	51.6	0.43
lac128	82684	26.7	15.9	**1.67**
r5nd1kem0d	89500	28.9	16.7	**1.73**
newhope512cca	109040	35.2	15.0	**2.34**
r5nd1kem5d	122492	39.5	13.8	**2.85**
frodokem640shake	4529184	1,461.0	1,223.0	**1.19**
Level 3				
ntruhps4096821	43100	13.9	48.4	0.29
sntrup761	48780	15.7	55.5	0.28
ntrulpr761	72372	23.3	59.6	0.39
kyber768	74040	23.9	17.9	**1.34**
saber2	115948	37.4	56.9	0.66
lac192	158628	51.2	21.4	**2.39**
r5nd3kem5d	209572	67.6	19.2	**3.52**
r5nd3kem0d	317244	102.3	21.9	**4.67**
frodokem976shake	9467152	3,053.9	1,642.5	**1.86**
Level 4 & 5				
sntrup857	60668	19.6	63.4	0.31
ntrulpr857	91416	29.5	67.3	0.44
kyber1024	103936	33.5	22.1	**1.52**
firesaber2	175844	56.7	65.2	0.87
lac256	188244	60.7	23.8	**2.55**
newhope1024cca	201772	65.1	21.3	**3.06**
r5nd5kem5d	368004	118.7	26.0	**4.57**
r5nd5kem0d	392492	126.6	29.2	**4.34**
frodokem1344shake	16379980	5,283.9	2,186.2	**2.42**
Decapsulation				
Level 1 & 2				
kyber512	37600	12.1	17.1	0.71
r5nd1kem0d	43000	13.9	19.3	0.72
sntrup653	59324	19.1	66.9	0.29
ntruhps2048677	62004	20.0	95.3	0.21
r5nd1kem5d	63624	20.5	15.7	**1.31**
ntruhrss701	63632	20.5	135.6	0.15
lightsaber2	69508	22.4	52.5	0.43
ntrulpr653	82732	26.7	70.9	0.38
lac128	105388	34.0	17.1	**1.99**
newhope512cca	109728	35.4	16.1	**2.19**
frodokem640shake	4494652	1,449.9	1,321.3	**1.10**
Level 3				
sntrup761	59120	19.1	78.9	0.24
kyber768	63916	20.6	20.1	**1.03**
ntruhps4096821	79448	25.6	107.1	0.24
ntrulpr761	85908	27.7	84.1	0.33
r5nd3kem5d	117028	37.8	22.8	**1.65**
saber2	118848	38.3	64.7	0.59
r5nd3kem0d	156692	50.5	27.0	**1.87**
lac192	243008	78.4	23.7	**3.30**
frodokem976shake	9380108	3,025.8	1,866.2	**1.62**
Level 4 & 5				
sntrup857	80904	26.1	86.8	0.30
kyber1024	91628	29.6	24.7	**1.20**
ntrulpr857	112116	36.2	97.5	0.37
firesaber2	182136	58.8	77.1	0.76
r5nd5kem0d	193228	62.3	35.9	**1.73**
newhope1024cca	206248	66.5	24.8	**2.68**
r5nd5kem5d	209136	67.5	31.7	**2.13**
lac256	377784	121.9	26.9	**4.54**
frodokem1344shake	16312844	5,262.2	3,119.9	**1.69**

portions of our implementations are written in portable C and run on a much less powerful processor, ARM Cortex-A53, at the frequency of 1.2 GHz. Hardware portions run in the programmable logic of Zynq UltraScale+, at a frequency specific to each algorithm, listed in Table 23, varying between 200 MHz for NTRU-HPS and NTRU-HRSS, through 322 MHz for Saber, until 490 MHz for NewHope. Even the frequency of NewHope is over 6 times smaller than the frequency of Intel Xeon. Additionally, all compared software implementations are optimized using AVX2 instructions, which let them take advantage of the parallelism present in each algorithm.

Under these circumstances, it is no surprise that Zynq UltraScale+ can outperform Intel Xeon only when its software/hardware implementation is fully optimized by moving all operations other than `randombytes()` to programmable logic. Such implementations of Kyber, LAC, NewHope, and Round5, outperform the best software implementations for both encapsulation and decapsulation. For encapsulation, the speed-ups vary from 1.34 for Kyber at level 3 to 4.67 for Round5 without error correction for level 3. For decapsulation, the speed-ups vary from 1.03 for Kyber at level 3 to 4.54 for LAC at level 5. The only exceptions are: Kyber at level 1, which reaches only the ratio of 0.94 for encapsulation, and 0.71 for decapsulation, and Round5 without error correction, which reaches only the ratio of 0.72 for level 1.

Somewhat surprisingly, also our software/hardware implementation of FrodoKEM outperforms the best software implementation, even though the percentage of operations offloaded to hardware in FrodoKEM is the smallest among all implemented KEMs, as shown in Figs. 9 and 10.

7 Conclusions

In this paper, we first reviewed the previous work on hardware and software/hardware implementations of Round 2 PQC schemes. Out of 26 candidates, six - NewHope, CRYSTALS-Kyber, FrodoKEM, Saber, Round5, and SIKE - received the highest coverage in terms of the number of implementations and related publications. All of them have both high-speed and lightweight implementations reported. Candidates with the Register-Transfer Level (RTL) high-speed implementations and no lightweight implementations include LAC, Classic McEliece, Picnic, and Rainbow. The publications on BIKE focused on key generation and decoding but did not report results for the entire KEM or PKE. Candidates with at least one software/hardware lightweight implementation but no RTL high-speed implementations include LEDAcrypt, CRYSTALS-DILITHIUM, and qTESLA. The coverage of the following candidates was limited to High-Level Synthesis implementations: SPHINCS+ and MQDSS. We are not aware of any publications on hardware or software/hardware implementations of Three Bears, HQC, NTS-KEM (before the merger with Classic McEliece), ROLLO, RQC, FALCON, GeMSS, and LUOV. With a few exceptions, the majority of lightweight implementations were software/hardware implementations based on RISC-V. The lattice-based family received by far the most extensive coverage. The following candidates from other families were shown competitive to lattice-based cryptography in terms of speed: for encryption and key exchange: Classic McEliece, for digital signatures Picnic and Rainbow. However, all of them were investigated primarily from the point of view of high-speed implementations.

In terms of the comparison of the lattice-based schemes, the previous publications were somewhat inconclusive. The largest differences were demonstrated in studies targeting ASIC implementations. These studies indicated the significant advantage of Kyber and NewHope over LAC and FrodoKEM, in terms of both the execution times of encapsulation and decapsulation, as well as power consumption and energy usage. However, the advantage over LAC could have been caused by devoting more effort and silicon area to the implementation of NTT (which benefits only NewHope and Kyber) vs. the implementation of vector arithmetic, units used to speed up LAC. The benchmarking of lattice-based signature schemes was limited to CRYSTALS-DILITHIUM and qTESLA. The conclusions were complicated by the withdrawal of heuristic parameters of qTESLA by the submitters on Aug. 20, 2019, and very limited coverage of the remaining parameter sets.

Due to the timing constraints, in our study, we decided to focus on 12 CCA-secure Key Encapsulation Mechanisms (KEMs) representing 8 out of 9 lattice-based key exchange schemes (all except Three Bears). Taking into account that even for this subset of candidates, the development of full RTL implementations appeared to be beyond the capabilities of a single group, we investigated the use of two techniques to speed up the development process: software/hardware co-design and High-Level Synthesis. A hybrid of these two approaches, with some modifications to the traditional HLS methodology, appeared to give quite promising results. However, we eventually devoted most of our effort to software/hardware co-design based on the merger of the RTL HDL code and optimized C code.

Unlike other groups, we applied software/hardware co-design to high-speed rather than lightweight implementations, which led to the choice of Xilinx Zynq UltraScale+, a state-of-the-art SoC FPGA family, as our primary platform. The differentiating factor is that this platform includes a hardwired ARM Cortex-A53 processor operating at the frequency of 1.2 GHz and a significant amount of programmable logic supporting hardware accelerators operating at the clock frequencies up to 500 MHz. Still, our designs remained almost completely portable due to leaving the software portion in C and modeling hardware portion in hardware description languages, such as VHDL, Verilog, and Chisel.

The detailed design methodology is described in this paper, and the corresponding code required to build a generic benchmarking platform, suitable for performing timing measurements of hardware and software/hardware co-designs is available for other groups to adopt. It is also our intention to make our implementations of PQC candidates open-source after the corresponding publications are accepted to peer-reviewed conferences or journals.

Our software/hardware co-design approach was successfully applied to all 12 mentioned above KEMs. For each KEM, multiple parameters sets, typically corresponding to three security levels, were supported. In order to determine FPGA resources required for each parameter set individually, the choice between parameter sets is performed during logic synthesis rather than at the run time.

For all algorithms other than FrodoKEM, the percentage of the original execution time in software taken by operations offloaded to hardware exceeded 97.4% for decapsulation and 96.9% for encapsulation. For FrodoKEM operations taking at least 94% of the execution time in software were offloaded to hardware. Significant speed-ups ranging between 7.6 and 111.1 were obtained versus a portable implementation in C, running on ARM Cortex-A53. More importantly, even when four KEMs representing three candidates - NewHope, NTRU, and Saber - were optimized in software by using NEON intrinsics, corresponding to special SIMD instructions of ARM, our sofware/hardware implementations maintained the lead by a factor varying between 1.81 for encapsulation in NTRU-HRSS up to 16.27 for encapsulation in NewHope. Finally, as an ultimate test, our implementations were compared with the software implementations optimized using AVX2 vector instructions, running on Intel Xeon E3-1220 v3, with the frequency 3.1 GHz. For each security level, between 4 and 6 software/hardware implementations, running on Zynq UltraScale+ with the 1.2 GHz ARM core were superior than the corresponding AVX2 implementations.

For each candidate, an attempt was made to offload as many as possible operations to hardware. For 50% of investigated KEMs, this percentage reached 100%. Thus, the corresponding implementations could be treated as hardware implementations, assuming that a random seed (of the size of 16, 24, or 32 bytes) was transferred to the hardware module during encapsulation. KEMs implemented using this approach included Kyber, LAC (v3a and v3b), NewHope, and Round5 (with and without error-correcting code).

Their code was benchmarked using Artix-7 and Virtex-7 FPGAs.

In terms of both the execution times and resource utilization, Round5 with an error-correcting code (R5ND_5d) outperformed Round5 without an error-correcting code (R5ND_0d). Similarly, LAC-v3b appeared superior over LAC-v3a in terms of both speed and use of FPGA resources. Then, when the best representatives of four candidates - Kyber, LAC, NewHope, and Round5 - were compared, the following conclusions could be drawn. The execution times of these candidates were extremely close to one another. For encapsulation, the execution times were within 10% from one another at the security level 5, within 22% at the security level 3, and within 32% at the security level 1. For decapsulation, the largest differences were 26% at level 5, 22% at level 3, and 48% at level 1. In multiple instances, just a change of an FPGA family from low-cost Artix-7 to high-performance Virtex-7 caused a significant change in the rankings, even though the HDL code remained exactly the same. As a result, we must conclude that the differences among these candidates in terms of speed are too small to give preference to any particular candidate. These results contradict one of the earlier reports placing LAC well behind NewHope and Kyber.

In terms of resource utilization, a small advantage belongs to NewHope and Kyber. Both of them use fewer LUTs and flip-flops than LAC and Round5, and their use of DSP units and BRAMs, although slightly higher, is very moderate. Additionally, both NewHope and Kyber use almost the same amount of resources independently of the security level. In the case of both LAC and Round5, resource usage increases sharply with the increase in security level. The former property appears to be an advantage for applications requiring support for the highest or all security levels. In particular, the k-in-1 designs, which support all *k* security levels and allow modifying them at run time, typically have only slightly higher resource utilization than that for the maximum security level. Thus, the flat dependence of the resource utilization on the security level implies a potential for very cost-effective k-in-1 designs. At the same time, this potential should still be confirmed through complete designs.

Independently of our own results, the results reported in [62] and summarized in Table 8 indicate that Saber–when implemented entirely in hardware–can reach the same performance level as the KEMs ranked on the first six positions in our study. It may also require fewer resources than at least Round5. With its resource utilization almost independent of the security level, it naturally allows 2-in-1 and 3-in-1 designs, supporting multiple security levels with a relatively small overhead in terms of speed and resource usage as compared to the implementation for the lowest security level. Similarly, estimates reported in [33] indicate that the performance of ThreeBears is likely to be very similar to that of Kyber and NewHope.

For the remaining 5 KEMs, representing FrodoKEM, NTRU, and NTRU Prime, the conclusions could be drawn only by comparing their software/hardware implementations and contrasting them with the corresponding software/hardware implementations of Kyber, LAC, NewHope, and Round5. In this case, all KEMs were implemented in Zynq UltraScale+. Hardware accelerators were assumed to be preloaded with appropriate public and private keys. Encapsulation started from generating 16-32 random bytes in software and passing these bytes to the hardware accelerator. Decapsulation started by sending the ciphertext to the hardware accelerator. Both operations ended when the shared secret was available in the memory of the processor core. Our evaluation revealed that FrodoKEM was by at least an order of magnitude slower than the remaining investigated KEMs. Ranking of the remaining candidates in hardware could not be determined conclusively based on their software/hardware co-design rankings. Software/hardware co-designs of Saber, NTRU-HRSS, and NTRU-HPS in particular, and somewhat less likely of Streamlined NTRU Prime and NTRU LPRime, could be possibly still significantly improved by offloading more operations to hardware, up to the level of bypassing at least one of the first six candidates in the ranking.

This pitfall of software/hardware co-designs was identified early on during the benchmarking process. It could have been overcome only if candidates were significantly different from the point of view of their hardware efficiency. Such large differences were not identified in the case of the mentioned above five lattice-based KEMs. Consequently, the only way to overcome this inherent weakness of the software/hardware methodology, when applied to this particular set of candidates, is to move all (or almost all) remaining operations of these algorithms to hardware. Doing that is, however, impractical at this point due to the timeline imposed by NIST.

At the same time, taking into account that moving more operations of these KEMs to hardware can only increase the resource usage of the corresponding hardware accelerators, it is still fair to compare their resource utilization with those of Kyber and NewHope. For NTRU-HPS and NTRU-HRSS, the concern is a large number of DSP units, exceeding 700 for NTRU-HRSS and 800 for NTRU-HPS at the security level 3. For Streamlined NTRU Prime and NTRU LPRime, the only concern is a relatively large number of LUTs, clearly exceeding that of LAC-v3b and approaching or exceeding that of Round5 with an error-correcting code (R5ND_5d).

Still, it is up to NIST and the cryptographic community to decide whether such relatively small differences in the hardware efficiency of lattice-based candidates should play any role in the Round 3 down-selection process.

8 Future Work

Future work will depend on the number and type of candidates qualified for Round 3. Based on the lessons learned from Round 2, the following adjustments may be advisable:

- More focus on hardware implementations vs. software/hardware implementations. Software/hardware implementations may still be helpful for lightweight implementations with a clear resource utilization threshold. In these implementations, moving more operations to hardware may be prohibited by exceeding the resource budget.

- More focus on comparisons across families, rather than within the same family. Round 2 designs illustrate substantial similarities between candidates belonging to the same family but give a hint of more profound differences among representatives of different families.

- More hardware platforms to focus on. The larger the spectrum of platforms, the higher certainty that the reported rankings are not artifacts of a particular platform and will carry over to future generations of integrated circuits. For FPGAs and SoC FPGAs, benchmarking should target families of at least two major vendors, Xilinx and Intel. For ASIC implementations, different standard-cell libraries should be considered. ASIC studies are particularly challenging, as they are more time-consuming and costlier. However, they are indispensable as they may lead to different conclusions than those obtained from FPGA investigations.

- More work on optimized software implementations targeting vector instructions of embedded processors, such as RISC-V and ARM (including NEON instructions).

- Investigation of lightweight implementations protected against side-channel and fault attacks should be conducted by multiple groups, serving interchangeably as attackers and defenders.

- Trade-offs among speed, area, power, energy, and resistance against side channel attacks should be thoroughly studied, especially for lightweight implementations.

References

[1] Erdem Alkim et al. *ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V*. Tech. rep. 049. 2020.

[2] Erdem Alkim et al. *NewHope - Algorithm Specifications and Supporting Documentation Version 1.1*. en. Tech. rep. Apr. 2020.

[3] Michał Andrzejczak. "The Low-Area FPGA Design for the Post-Quantum Cryptography Proposal Round5". In: *2019 Federated Conference on Computer Science and Information Systems*. Vol. 18. Leipzig, Germany, Sept. 2019, pp. 213–219. DOI: `10/ggbsbd`.

[4] Michal Andrzejczak, Farnoud Farahmand, and Kris Gaj. "Full Hardware Implementation of the Post-Quantum Public-Key Cryptography Scheme Round5". en. In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun, Mexico: IEEE, Dec. 2019, pp. 1–2. ISBN: 978-1-72811-957-1. DOI: `10.1109/ReConFig48160.2019.8994765`.

[5] Nicolas Aragon. *BIKE: Bit Flipping Key Encapsulation*. en. Tech. rep. May 2020, p. 40.

[6] ARM. *Neon Intrinsics Reference*. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics. 2020.

[7] ARM. *Neon Programmer's Guide for Armv8-A*. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/neon-programmers-guide-for-armv8-a. 2020.

[8] Roberto Avanzi et al. *KYBER: Algorithm Specications And Supporting Documentation, Version 2.0, NIST PQC Round 2*. en. Tech. rep. Apr. 2019, p. 37.

[9] Reza Azarderakhsh et al. "Key Compression for Isogeny-Based Cryptosystems". en. In: *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography - AsiaPKC '16*. Xi'an, China: ACM Press, 2016, pp. 1–10. ISBN: 978-1-4503-4286-5. DOI: `10/ggbsbz`.

[10] Reza Azarderakhsh et al. *NEON PQCryto: Fast and Parallel Ring-LWE Encryption on ARM NEON Architecture*. Cryptology ePrint Archive 2015/1081. Nov. 2015.

[11] Brian Baldwin et al. "FPGA Implementations of the Round Two SHA-3 Candidates". In: *2010 International Conference on Field Programmable Logic and Applications, FPL 2010*. Milan, Italy, Aug. 2010, pp. 400–407. ISBN: 978-1-4244-7842-2. DOI: `10/bmn2zv`.

[12] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-Based Protocols". en. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.4 (Aug. 2019). DOI: `10.13154/tches.v2019.i4.17-61`.

[13] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. *Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-Based Protocols (Extended Version)*. Cryptology ePrint Archive 2019/1140. Oct. 2019.

[14] Kanad Basu et al. *NIST Post-Quantum Cryptography- A Hardware Evaluation Study*. Cryptology ePrint Archive 2019/047. May 2019.

[15] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. https://bench.cr.yp.to. 2019.

[16] Daniel J Bernstein and Peter Schwabe. "NEON Crypto". In: *Cryptographic Hardware and Embedded Systems - CHES 2012*. Vol. 7428. LNCS. Leuven, Belgium, Sept. 2012, pp. 320–339. DOI: `10.1007/978-3-642-33027-8_19`.

[17] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness - Web Page.* https://competitions.cr.yp.to/caesar.html. 2019.

[18] Cryptographic Engineering Research Group (CERG) at George Mason University. *Hardware Benchmarking of CAESAR Candidates.* https://cryptography.gmu.edu/athena/index.php?id=CAES 2019.

[19] Viet B Dang et al. "Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign". In: *2019 International Conference on Field Programmable Technology, FPT 2019.* Tianjin, China: IEEE, Dec. 9-13, 2019, pp. 206–214. DOI: 10.1109/ICFPT47387.2019.00032.

[20] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. *Efficient and Fast Hardware Architectures for SIKE Round 2 on FPGA.* Cryptology ePrint Archive 2020/611. May 2020.

[21] Farnoud Farahmand. *Benchmarking Setup for Software/Hardware Implementations of PQC Schemes.* Sept. 2019.

[22] Farnoud Farahmand et al. "Evaluating the Potential for Hardware Acceleration of Four NTRU-Based Key Encapsulation Mechanisms Using Software/Hardware Codesign". In: *10th International Conference on Post-Quantum Cryptography, PQCrypto 2019.* LNCS. Chongqing, China: Springer, May 2019.

[23] Farnoud Farahmand et al. "Minerva: Automated Hardware Optimization Tool". In: *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017.* Cancun: IEEE, Dec. 2017, pp. 1–8.

[24] Farnoud Farahmand et al. "Software/Hardware Codesign of the Post Quantum Cryptography Algorithm NTRUEncrypt Using High-Level Synthesis and Register-Transfer Level Design Methodologies". In: *29th International Conference on Field Programmable Logic and Applications, FPL 2019.* Barcelona, Spain: IEEE, Sept. 2019, pp. 225–231. ISBN: 978-1-72814-884-7. DOI: 10.1109/FPL.2019.00042.

[25] Ahmed Ferozpuri and Kris Gaj. "High-Speed FPGA Implementation of the NIST Round 1 Rainbow Signature Scheme". In: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig).* Cancun, Mexico: IEEE, Dec. 2018, pp. 1–8. ISBN: 978-1-72811-968-7. DOI: 10/ggbsdm.

[26] Ahmed Ferozpuri et al. *Hardware API for Post-Quantum Public Key Cryptosystems.* GMU Report. Fairfax, VA: George Mason University, Apr. 2018.

[27] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. *RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography.* Tech. rep. 446. 2020.

[28] Tim Fritzmann et al. "Towards Reliable and Secure Post-Quantum Co-Processors Based on RISC-V". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE).* ZSCC: 0000001. Mar. 2019, pp. 1148–1153. DOI: 10.23919/DATE.2019.8715173.

[29] Kris Gaj. "Challenges and Rewards of Implementing and Benchmarking Post-Quantum Cryptography in Hardware". In: *2018 Great Lakes Symposium on VLSI, GLSVLSI 2018.* Chicago, IL, USA: ACM Press, 2018, pp. 359–364. ISBN: 978-1-4503-5724-1. DOI: 10/ggbscs.

[30] Kris Gaj, Ekawat Homsirikamol, and Marcin Rogawski. "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs". In: *Cryptographic Hardware and Embedded Systems, CHES 2010.* Vol. 6225. LNCS. Santa Barbara, CA, Aug. 2010, pp. 264–278. ISBN: 978-3-642-15030-2 978-3-642-15031-9. DOI: 10.1007/978-3-642-15031-9_18.

[31] Kris Gaj et al. “ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs”. In: *2010 International Conference on Field Programmable Logic and Applications, FPL 2010*. Milan, Italy: IEEE, Aug. 2010, pp. 414–421. ISBN: 978-1-4244-7842-2. DOI: 10/d2bzw2.

[32] Kris Gaj et al. *Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs*. Cryptology ePrint Archive 2012/368. 2012.

[33] Mike Hamburg. *ThreeBears Estimates on ASIC*. June 2020.

[34] E. Homsirikamol et al. *Implementer’s Guide to Hardware Implementations Compliant with the CAESAR Hardware API*. GMU Report. Fairfax, VA: GMU, 2016.

[35] Ekawat Homsirikamol and Kris Gaj. “Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study”. In: *Applied Reconfigurable Computing - ARC 2015*. Vol. 9040. LNCS. Cham: Springer International Publishing, 2015, pp. 217–228. ISBN: 978-3-319-16213-3 978-3-319-16214-0. DOI: 10.1007/978-3-319-16214-0_18.

[36] Ekawat Homsirikamol and Kris Gaj. “Toward a New HLS-Based Methodology for FPGA Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study”. In: *2017 International Conference on Field Programmable Technology, FPT 2017*. Melbourne, Australia: IEEE, Dec. 2017, pp. 120–127. ISBN: 978-1-5386-2656-6. DOI: 10/ggbsf4.

[37] Ekawat Homsirikamol, Panasayya Yalla, and Farnoud Farahmand. *Development Package for Hardware Implementations Compliant with the CAESAR Hardware API*. https://cryptography.gmu.edu/athena/index.php?id=CAESAR. 2016.

[38] Ekawat Homsirikamol et al. *CAESAR Hardware API*. Cryptology ePrint Archive 2016/626. 2016.

[39] James Howe. “Optimised Lattice-Based Key Encapsulation in Hardware”. en. In: *Second NIST Post-Quantum Cryptography Standardization Conference 2019*. Aug. 2019, p. 13.

[40] Jingwei Hu et al. “Lightweight Key Encapsulation Using LDPC Codes on FPGAs”. en. In: *IEEE Trans. Comput.* (2019). ISSN: 0018-9340, 1557-9956, 2326-3814. DOI: 10.1109/TC.2019.2948323.

[41] Arpan Jati et al. *SPQCop: Side-Channel Protected Post-Quantum Cryptoprocessor*. Cryptology ePrint Archive 2019/765. June 2019.

[42] Daniel Kales et al. “Efficient FPGA Implementations of LowMC and Picnic”. In: *The Cryptographers’ Track at the RSA Conference 2020, CT-RSA 2020*. San Francisco: Springer, Feb. 2020.

[43] Matthias J. Kannwischer et al. *Pqm4 - Post-Quantum Crypto Library for the {ARM} {Cortex-M4}*. https://github.com/mupq/pqm4. 2019.

[44] Jens-Peter Kaps et al. “Lightweight Implementations of SHA-3 Candidates on FPGAs”. In: *12th International Conference on Cryptology in India, Indocrypt 2011*. Vol. 7107. LNCS. Chennai, India, Dec. 2011, pp. 270–289. ISBN: 978-3-642-25577-9 978-3-642-25578-6. DOI: 10.1007/978-3-642-25578-6_20.

[45] Miroslav Knezevic et al. “Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.5 (May 2012), pp. 827–840. ISSN: 1063-8210, 1557-9999. DOI: 10/ctjzhr.

[46] B. Koziel, R. Azarderakhsh, and M. M. Kermani. “A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography”. In: *IEEE Transactions on Computers* 67.11 (Nov. 2018), pp. 1594–1609. ISSN: 0018-9340. DOI: 10/gff4vv.

[47] Brian Koziel et al. “NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM”. en. In: *Cryptology and Network Security*. Ed. by Sara Foresti and Giuseppe Persiano. Vol. 10052. Cham: Springer International Publishing, 2016, pp. 88–103. ISBN: 978-3-319-48964-3 978-3-319-48965-0. DOI: 10.1007/978-3-319-48965-0_6.

[48] Brian Koziel et al. “Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.1 (Jan. 2017), pp. 86–99. ISSN: 1549-8328, 1558-0806. DOI: 10/gd89pp.

[49] Brian Koziel et al. *SIKE’d Up: Fast and Secure Hardware Architectures for Supersingular Isogeny Key Encapsulation*. en. Cryptology ePrint Archive 2019/711. June 2019, p. 27.

[50] Weiqiang Liu et al. “High Performance Modular Multiplication for SIDH”. en. In: *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* (2019), pp. 1–1. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2019.2960330.

[51] Patrick Longa. *FourQNEON: Faster Elliptic Curve Scalar Multiplications on ARM Processors*. en. Cryptology ePrint Archive 2016/645. July 2016, p. 16.

[52] Xianhui Lu et al. *LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus*. Cryptology ePrint Archive 2018/1009. Dec. 2019.

[53] Pedro Maat C. Massolino et al. “A Compact and Scalable Hardware/Software Co-Design of SIKE”. en. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Mar. 2020), pp. 245–271. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i2.245-271.

[54] Jose Maria Bermudo Mera et al. *Compact Domain-Specific Co-Processor for Accelerating Module Lattice-Based Key Encapsulation Mechanism*. en. Cryptology ePrint Archive 2020/321. Mar. 2020, p. 15.

[55] Richard Newell. *Survey of Notable Security-Enhancing Activities in the RISC-V Universe*. 17th International Workshop on Cryptographic Architectures Embedded in Logic Devices, CryptArchi 2019. Pruhonice, Czech Republic, June 2019.

[56] Duc Tri Nguyen, Viet B. Dang, and Kris Gaj. “A High-Level Synthesis Approach to the Software/Hardware Codesign of NTT-Based Post-Quantum Cryptography Algorithms”. en. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. Tianjin, China: IEEE, Dec. 2019, pp. 371–374. ISBN: 978-1-72812-943-3. DOI: 10.1109/ICFPT47387.2019.00070.

[57] Duc Tri Nguyen, Viet B Dang, and Kris Gaj. “High-Level Synthesis in Implementing and Benchmarking Number Theoretic Transform in Lattice-Based Post-Quantum Cryptography Using Software/Hardware Codesign”. In: *16th International Symposium on Applied Reconfigurable Computing, ARC 2020*. Apr. 2020.

[58] NIST. *PQC - API Notes*. 2017.

[59] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Book version: 0.0.1. Strawberry Canyon LLC, Oct. 2017.

[60] Andrew H Reinders et al. *Efficient BIKE Hardware Design with Constant-Time Decoder*. Cryptology ePrint Archive 2020/117. Feb. 2020.

[61] Vincent Rijmen, Antoon Bosselaers, and Paulo Barreto. *Optimized ANSI C Code for the Rijndael Cipher (Now AES), Rijndael-Alg-Fst.c, v3.0*. Dec. 2000.

[62] Sujoy Sinha Roy and Andrea Basso. *High-Speed Instruction-Set Coprocessor for Lattice-Based Key Encapsulation Mechanism: Saber in Hardware*. Cryptology ePrint Archive 2020/434. Apr. 2020.

[63] Markku-Juhani O. Saarinen. *Pqcbench*. https://github.com/mjosaarinen/pqcbench. 2019.

[64] Hwajeong Seo et al. “Montgomery Modular Multiplication on ARM-NEON Revisited”. In: *International Conference on Information Security and Cryptology - ICISC 2014*. Vol. 8949. LNCS. Cham: Springer International Publishing, 2014, pp. 328–342. ISBN: 978-3-319-15942-3 978-3-319-15943-0. DOI: `10.1007/978-3-319-15943-0`.

[65] Hwajeong Seo et al. *Parallel Implementation of Number Theoretic Transform*. en. Cryptology ePrint Archive 2015/1024. Oct. 2015, p. 12.

[66] Hwajeong Seo et al. *SIDH on ARM: Faster Modular Multiplications for Faster Post-Quantum Supersingular Isogeny Key Exchange*. Cryptology ePrint Archive 2018/700. July 2018.

[67] Douglas Stebila and Michele Mosca. *Liboqs - Master Branch*. https://github.com/open-quantum-safe/liboqs. 2019.

[68] Silvan Streit and Fabrizio De Santis. “Post-Quantum Key Exchange on ARMv8-A: A New Hope for NEON Made Simple”. en. In: *IEEE Transactions on Computers* 67.11 (Nov. 2018), pp. 1651–1662. ISSN: 0018-9340, 1557-9956, 2326-3814. DOI: `10/gff3sc`.

[69] FrodoKEM Submission Team. *Round 2 Submissions - FrodoKEM Candidate Submission Package*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions. Apr. 2019.

[70] NTRU Prime Submission Team. *Round 2 Submissions - NTRU Prime Candidate Submission Package*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions. Apr. 2019.

[71] NTRU Submission Team. *Round 2 Submissions - NTRU Candidate Submission Package*. Apr. 2019.

[72] Round5 Submission Team. *Round 2 Submissions - Round5 Candidate Submission Package*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions. Apr. 2019.

[73] Saber Submission Team. *Round 2 Submissions - Saber Candidate Submission Package*. Apr. 2019.

[74] Wen Wang, Jakub Szefer, and Ruben Niederhagen. “FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes”. In: *9th International Conference on Post-Quantum Cryptography, PQCrypto 2018*. Ed. by Tanja Lange and Rainer Steinwandt. Vol. 10786. LNCS. Fort Lauderdale, Florida: Springer International Publishing, Apr. 2018, pp. 77–98. ISBN: 978-3-319-79062-6 978-3-319-79063-3. DOI: `10.1007/978-3-319-79063-3_4`.

[75] Wen Wang et al. *Parameterized Hardware Accelerators for Lattice-Based Cryptography and Their Application to the HW/SW Co-Design of qTESLA*. Cryptology ePrint Archive 2020/054. Apr. 2020.

[76] Wen Wang et al. “XMSS and Embedded Systems - XMSS Hardware Accelerators for RISC-V”. In: *Selected Areas in Cryptography – SAC 2019*. Vol. 11959. LNCS. Waterloo, Ontario, Canada: Springer, 2019, pp. 523–550.

[77] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual . Volume I: Unprivileged ISA v2.2*. Tech. rep. 20190608-Base-Ratified. June 2019, p. 236.

[78] Andrew Waterman and Krste Asanovic. “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, v1.12”. In: (June 2019), p. 113.

[79] Guozhu Xin et al. “VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2020), pp. 1–13. ISSN: 1558-0806. DOI: `10.1109/TCSI.2020.2983185`.

[80] Neng Zhang et al. “Highly Efficient Architecture of NewHope-NIST on FPGA Using Low-Complexity NTT/INTT”. en. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Mar. 2020), pp. 49–72. ISSN: 2569-2925. DOI: `10.13154/tches.v2020.i2.49-72`.

A Results of Profiling

Table 29: Results of profiling FrodoKEM

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
FrodoKEM1344 - Encapsulation					
1. frodo_mul_add_sa_plus_e	58,577.48	94.36	**1.1 frodo_mul_add_sa_plus_e**	1,328.39	60.76
2. Shake128 and frodo_sample_n x3	1,416.27	2.28	**1.2 Shake128 and frodo_sample_n**		
3. frodo_mul_add_sb_plus_e	654.64	1.05	**1.3 frodo_mul_add_sb_plus_e**		
4. Shake256	569.60	0.92	**1.4 Shake256**		
5. frodo_pack	386.22	0.62	2. frodo_pack	386.22	17.67
6. frodo_unpack	276.00	0.44	3. frodo_unpack	276.00	12.62
Others	195.62	0.32	Others	195.62	8.95
Total	62,075.83	100.00	Total	2,186.23	100.00
FrodoKEM1344 - Decapsulation					
1. frodo_mul_add_sa_plus_e	58,754.02	94.19	**1.1 frodo_mul_add_sa_plus_e**	1,316.52	42.20
2. Shake128 and frodo_sample_n x3	883.14	1.42	**1.2 Shake128 and frodo_sample_n**		
3. frodo_unpack x3	765.56	1.23	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_mul_add_sb_plus_e	649.68	1.04	**1.4 Shake256**		
5. frodo_mul_bs	507.08	0.81	2. frodo_unpack x3	765.56	24.54
6. Shake256	286.64	0.46	3. frodo_mul_bs	507.08	16.25
Others	530.74	0.85	Others	530.74	17.01
Total	62,376.86	100.00	Total	3,119.90	100.00
FrodoKEM976 - Encapsulation					
1. frodo_mul_add_sa_plus_e	31,430.38	90.82	**1.1 frodo_mul_add_sa_plus_e**	760.74	46.32
2. Shake128 and frodo_sample_n x3	1,410.18	4.07	**1.2 Shake128 and frodo_sample_n**		
3. frodo_mul_add_sb_plus_e	472.16	1.36	**1.3 frodo_mul_add_sb_plus_e**		
4. Shake256	414.11	1.20	**1.4 Shake256**		
5. frodo_pack	357.58	1.03	2. frodo_pack	357.58	21.77
6. frodo_unpack	297.73	0.86	3. frodo_unpack	297.73	18.13
Others	226.40	0.65	Others	226.40	13.78
Total	34,608.54	100.00	Total	1,642.45	100.00
FrodoKEM976 - Decapsulation					
1. frodo_mul_add_sa_plus_e	31,441.14	90.74	**1.1 frodo_mul_add_sa_plus_e**	749.76	40.18
2. Shake128 and frodo_sample_n x3	1,410.86	4.07	**1.2 Shake128 and frodo_sample_n**		
3. frodo_unpack x3	594.63	1.72	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_mul_add_sb_plus_e	471.29	1.36	**1.4 Shake256**		
5. frodo_mul_bs	368.32	1.06	2. frodo_unpack x3	594.63	31.86
6. Shake256	208.83	0.60	3. frodo_mul_bs	368.32	19.74
Others	153.51	0.44	Others	153.51	8.23
Total	34,648.58	100.00	Total	1,866.22	100.00
FrodoKEM640 - Encapsulation					
1. frodo_mul_add_sa_plus_e	13,794.27	85.19	**1.1 frodo_mul_add_sa_plus_e**	352.52	28.82
2. Shake128 and frodo_sample_n x3	1,002.40	6.19	**1.2 Shake128 and frodo_sample_n**		
3. frodo_mul_add_sb_plus_e	309.68	1.91	**1.3 frodo_mul_add_sb_plus_e**		
4. Shake256	215.55	1.33	**1.4 Shake256**		
5. frodo_pack	291.83	1.80	2. frodo_pack	291.83	23.86
6. frodo_unpack	277.26	1.71	3. frodo_unpack	277.26	22.67
Others	301.38	1.86	Others	301.38	24.64
Total	16,192.37	100.00	Total	1,222.99	100.00
FrodoKEM640 - Decapsulation					
1. frodo_mul_add_sa_plus_e	13,793.01	85.18	**1.1 frodo_mul_add_sa_plus_e**	342.95	25.95
2. Shake128 and frodo_sample_n x3	1,002.85	6.19	**1.2 Shake128 and frodo_sample_n**		
3. frodo_unpack x3	548.74	3.39	**1.3 frodo_mul_add_sb_plus_e**		
4. frodo_mul_add_sb_plus_e	309.21	1.91	**1.4 Shake256**		
5. frodo_mul_bs	242.40	1.50	2. frodo_unpack x3	548.74	41.53
6. Shake256	108.93	0.67	3. frodo_mul_bs	242.40	18.35
Others	187.23	1.16	Others	187.23	14.17
Total	16,192.37	100.00	Total	1,321.32	100.00

Table 30: Results of profiling Kyber

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
Kyber CCA-KEM 1024 - Encapsulation					
1. indcpa_enc	736.7	93.55	**1.1. indcpa_enc**	20.5	93.18
2. hash	49.3	6.26	**1.2. hash**		
3. randombytes	1.5	0.19	2. randombytes	1.5	6.82
Total	787.5	100.00	Total	22.0	100.00
Kyber CCA-KEM 1024 - Decapsulation					
1. indcpa_enc	734.2	76.99	**1.1 indcpa_enc**	24.7	100.00
2. indcpa_dec	191.7	20.10	**1.2 indcpa_dec**		
3. hash & verify	27.7	2.90	**1.3 hash & verify**		
Total	953.7	100.00	Total	24.7	100.00
Kyber CCA-KEM 768 - Encapsulation					
1. indcpa_enc	496.3	92.48	**1.1. indcpa_enc**	16.3	91.58
2. hash	38.9	7.24	**1.2. hash**		
3. randombytes	1.5	0.28	2. randombytes	1.5	8.42
Total	536.7	100.00	Total	17.8	100.00
Kyber CCA-KEM 768 - Decapsulation					
1. indcpa_enc	493.2	73.60	**1.1 indcpa_enc**	20.1	100.00
2. indcpa_dec	154.6	23.07	**1.2 indcpa_dec**		
3. hash & verify	22.3	3.33	**1.3 hash & verify**		
Total	670.1	100.00	Total	20.1	100.00
Kyber CCA-KEM 512 - Encapsulation					
1. indcpa_enc	302.8	91.19	**1.1. indcpa_enc**	13.7	90.10
2. hash	27.8	8.36	**1.2. hash**		
3. randombytes	1.5	0.45	2. randombytes	1.5	9.90
Total	332.0	100.00	Total	15.2	100.00
Kyber CCA-KEM 512 - Decapsulation					
1. indcpa_enc	298.5	68.93	**1.1 indcpa_enc**	17.1	100.00
2. indcpa_dec	117.9	27.22	**1.2 indcpa_dec**		
3. hash & verify	16.7	3.85	**1.3 hash & verify**		
Total	433.0	100.00	Total	17.1	100.00

Table 31: Results of profiling LAC-v3a

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
LAC-v3a-256 - Encapsulation					
1. pke_enc_seed	901.0	99.42	**1.1 pke_enc_seed**	22.3	93.69
2. hash_to_k	2.3	0.26	**1.2 hash_to_k**		
3. random_bytes	1.5	0.17	**1.3 Others**		
Others	1.4	0.16	2. random_bytes	1.5	6.31
Total	906.3	100.00	Total	23.8	100.00
LAC-v3a-256 - Decapsulation					
1. pke_enc_seed	901.1	65.37	**1.1 pke_enc_seed**	26.9	100.00
2. pke_dec	472.4	34.27	**1.2 pke_dec**		
3. hash_to_k	2.3	0.17	**1.3 hash_to_k**		
Others	2.6	0.19	**1.4 Others**		
Total	1,378.4	100.00	Total	26.9	100.00
LAC-v3a-192 - Encapsulation					
1. pke_enc_seed	558.5	99.06	**1.1 pke_enc_seed**	19.9	92.99
2. hash_to_k	2.3	0.41	**1.2 hash_to_k**		
3. random_bytes	1.5	0.27	**1.3 Others**		
Others	1.5	0.26	2. random_bytes	1.5	7.01
Total	563.8	100.00	Total	21.4	100.00
LAC-v3a-192 - Decapsulation					
1. pke_enc_seed	558.7	71.44	**1.1 pke_enc_seed**	23.7	100.00
2. pke_dec	218.6	27.95	**1.2 pke_dec**		
3. hash_to_k	2.3	0.30	**1.3 hash_to_k**		
Others	2.5	0.31	**1.4 Others**		
Total	782.1	100.00%	Total	23.7	100.00
LAC-v3a-128 - Encapsulation					
1. pke_enc_seed	328.6	98.73	**1.1 pke_enc_seed**	14.9	93.73
2. hash_to_k	2.3	0.69	**1.2 hash_to_k**		
3. random_bytes	1.0	0.30	**1.3 Others**		
Others	0.9	0.27	2. random_bytes	1.0	6.27
Total	332.8	100.00	Total	15.9	100.00
LAC-v3a-128 - Decapsulation					
1. pke_enc_seed	328.5	71.01	**1.1 pke_enc_seed**	17.1	100.00
2. pke_dec	130.3	28.17	**1.2 pke_dec**		
3. hash_to_k	2.3	0.50	**1.3 hash_to_k**		
Others	1.5	0.32	**1.4 Others**		
Total	462.6	100.00	Total	17.1	100.00

Table 32: Results of profiling LAC-v3b

Function	**Time** [us]	**Time** [%]	**Function**	**Time** [us]	**Time** [%]
Software			**Software/Hardware**		
LAC-v3b-256 - Encapsulation					
1. pke_enc_seed	861.8	99.05	**1.1 pke_enc_seed**	19.6	92.88
2. hash_to_k	2.3	0.27	**1.2 hash_to_k**		
3. random_bytes	1.5	0.17	**1.3 Others**		
Others	4.4	0.51	2. random_bytes	1.5	7.12
Total	870.0	100.00	Total	21.1	100.00
LAC-v3b-256 - Decapsulation					
1. pke_enc_seed	861.6	64.69	**1.1 pke_enc_seed**	24.1	100.00
2. pke_dec	463.3	34.78	**1.2 pke_dec**		
3. hash_to_k	2.3	0.17	**1.3 hash_to_k**		
Others	4.7	0.35	**1.4 Others**		
Total	1,331.9	100.00	Total	24.1	100.00
LAC-v3b-192 - Encapsulation					
1. pke_enc_seed	522.7	98.54	**1.1 pke_enc_seed**	16.8	91.82
2. hash_to_k	2.3	0.44	**1.2 hash_to_k**		
3. random_bytes	1.5	0.28	**1.3 Others**		
Others	3.9	0.74	2. random_bytes	1.5	8.18
Total	530.4	100.00	Total	18.3	100.00
LAC-v3b-192 - Decapsulation					
1. pke_enc_seed	522.8	70.59	**1.1 pke_enc_seed**	20.6	100.00
2. pke_dec	211.2	28.51	**1.2 pke_dec**		
3. hash_to_k	2.3	0.31	**1.3 hash_to_k**		
Others	4.4	0.59	**1.4 Others**		
Total	740.7	100.00	Total	20.6	100.00
LAC-v3b-128 - Encapsulation					
1. pke_enc_seed	309.4	98.34	**1.1 pke_enc_seed**	13.4	93.07
2. hash_to_k	2.3	0.73	**1.2 hash_to_k**		
3. random_bytes	1.0	0.32	**1.3 Others**		
Others	1.9	0.61	2. random_bytes	1.0	6.93
Total	314.6	100.00	Total	14.4	100.00
LAC-v3b-128 - Decapsulation					
1. pke_enc_seed	309.3	70.25	**1.1 pke_enc_seed**	15.5	100.00
2. pke_dec	126.3	28.69	**1.2 pke_dec**		
3. hash_to_k	2.3	0.52	**1.3 hash_to_k**		
Others	2.3	0.53	**1.4 Others**		
Total	440.2	100.00	Total	15.5	100.00

Table 33: Results of profiling NewHope

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
NewHope CCA-KEM 1024 - Encapsulation					
1. cpapke_enc	668.3	91.33	**1.1. cpapke_enc**	19.7	92.92
2. hash	62.0	8.47	**1.2. hash**		
3. randombytes	1.5	0.20	2. randombytes	1.5	7.08
Total	731.7	100.00	Total	21.2	100.00
NewHope CCA-KEM 1024 - Decapsulation					
1. cpapke_enc	660.7	74.01	**1.1 cpapke_enc**	24.8	100.00
2. cpapke_dec	193.9	21.72	**1.2 cpapke_dec**		
3. hash & verify	38.2	4.27	**1.3 hash & verify**		
Total	892.7	100.00	Total	24.8	100.00
NewHope CCA-KEM 512 - Encapsulation					
1. cpapke_enc	316.9	89.60	**1.1. cpapke_enc**	13.5	89.98
2. hash	35.3	9.97	**1.2. hash**		
3. randombytes	1.5	0.42	2. randombytes	1.5	10.02
Total	353.6	100.00	Total	15.0	100.00
NewHope CCA-KEM 512 - Decapsulation					
1. cpapke_enc	311.8	72.92	**1.1 cpapke_enc**	16.1	100.00
2. cpapke_dec	93.3	21.82	**1.2 cpapke_dec**		
3. hash & verify	22.5	5.26	**1.3 hash & verify**		
Total	427.5	100.00	Total	16.1	100.00

Table 34: Results from profiling NTRU

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
NTRU HPS 4096821- Encapsulation					
1. poly_Rq_mul	3,954.9	92.29	**1.1 poly_Rq_mul**	29.8	63.77
2. owcpa_samplemsg	251.4	5.87	**1.2 owcpa_samplemsg**		
3. shake256	54.3	1.27	**1.3 shake256**		
4. sha3_256	7.6	0.18	**1.4 sha3_256**		
Others	16.9	0.39	Others	16.9	36.23
Total	4,285.1	100.00	Total	46.7	100.00
NTRU HPS 4096821 - Decapsulation					
1. poly_S3_mul	3,972.1	33.15	**1.1 poly_Rq_mul**	37.4	34.92
2. poly_Sq_mul	3,960.3	33.05	**1.2 poly_S3_mul**		
3. poly_Rq_mul	3,955.4	33.01	**1.3 poly_Sq_mul**		
4. poly_S3_frombytes x2	31.1	0.26	**1.4 sha3_256 x2**		
5. sha3_256 x2	24.2	0.20	2. poly_S3_frombytes x2	31.1	29.01
Others	38.6	0.32	Others	38.6	36.07
Total	11,981.7	100.00	Total	107.1	100.00
NTRU HPS 2048677- Encapsulation					
1. poly_Rq_mul	2,692.6	90.92	**2.1 poly_Rq_mul**	26.0	62.44
2. owcpa_samplemsg	199.8	6.75	**2.2 owcpa_samplemsg**		
3. shake256	45.9	1.55	**2.3 shake256**		
4. sha3_256	7.5	0.25	**2.4 sha3_256**		
Others	15.6	0.53	Others	15.6	37.56
Total	2,961.5	100.00	Total	41.6	100.00
NTRU HPS 2048677 - Decapsulation					
1. poly_S3_mul	2,706.8	33.11	**1.1 poly_Rq_mul**	34.1	35.77
2. poly_Sq_mul	2,693.2	32.94	**1.2 poly_S3_mul**		
3. poly_Rq_mul	2,693.1	32.94	**1.3 poly_Sq_mul**		
4. poly_S3_frombytes x2	25.9	0.32	**1.4 sha3_256 x2**		
5. sha3_256	20.6	0.25	2. poly_S3_frombytes x2	25.9	27.14
Others	35.3	0.43	Others	35.3	37.10
Total	8,174.9	100.00	Total	95.3	100.00
NTRU-HRSS - Encapsulation					
1. poly_Rq_mul	2,886.1	97.36	1. poly_lift	27.9	42.74
2. shake256	24.2	0.82	**2.1 poly_Rq_mul**	22.3	34.12
3. poly_lift	27.9	0.94	**2.2 owcpa_samplemsg**		
4.sha3_256	7.6	0.26	**2.3 shake256**		
5. owcpa_samplemsg	3.5	0.12	**2.3 sha3_256**		
Others	15.1	0.51	Others	15.1	23.14
Total	2,964.5	100.00	Total	65.3	100.00
NTRU-HRSS - Decapsulation					
1. poly_S3_mul	2,900.8	33.00	**1.1 poly_Rq_mul**	46.4	34.17
2. poly_Sq_mul	2,890.7	32.89	**1.2 poly_S3_mul**		
3. poly_Rq_mul	2,886.6	32.84	**1.3 poly_Sq_mul**		
4. poly_lift	27.2	0.31	**1.4 sha3_256**		
5. sha3_256	22.3	0.25	2. poly_lift	27.2	20.05
Others	62.1	0.71	Others	62.1	45.78
Total	8,789.8	100.00	Total	135.6	100.00

Table 35: Results of profiling NTRULPRime

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
NTRULPrime857 - Encapsulation					
1. Rq_mult_small x2	1,448.9	70.32	**1.1 Short_fromlist**	69.6	70.97
2. Short_fromlist	261.6	12.69	**1.2 Expand x2**		
3. Expand x2	245.4	11.91	**1.3 Hash X4**		
4. Hash x4	60.7	2.95	**1.4 Rq_mult_small x2**		
5. crypto_decode_857x1723	28.5	1.38	**1.5.crypto_encode_857x1723round**		
6. crypto_encode_857x1723round	4.7	0.23	**1.6 Others**		
Others	10.9	0.53	2. crypto_decode_857x1723	28.5	29.03
Total	2,060.5	100.00	Total	98.0	100.00
NTRULPrime857 - Decapsulation					
1. Rq_mult_small x3	2,173.5	78.00	**1.1 Short_fromlist**	47.3	48.52
2. Short_fromlist	261.4	9.38	**1.2 expand x2**		
3. expand x2	246.5	8.85	**1.3 Rq_mult_small x3**		
4. crypto_decode_857x1723 x2	50.2	1.80	**1.4 Hash x3**		
5. Hash x3	34.0	1.22	**1.6 crypto_encode_857x1723round**		
6. crypto_encode_857x1723round	4.8	0.17	**1.7 Others**		
Others	16.3	0.58	2. crypto_decode_857x1723 x2	50.2	51.48
Total	2,786.6	100.00	Total	97.5	100.00
NTRULPrime761 - Encapsulation					
1. Rq_mult_small x2	1,169.5	68.62	**1.1 Short_fromlist**	66.0	67.07
2. Short_fromlist	226.2	13.27	**1.2 Expand x2**		
3. Expand x2	214.8	12.60	**1.3 Hash X4**		
4. Hash X4	54.4	3.19	**1.4 Rq_mult_small x2**		
5. crypto_decode_761x1531	25.7	1.51	**1.5 crypto_encode_761x1531round**		
6. crypto_encode_761x1531round	4.2	0.25	**1.6 Others**		
Others	9.5	0.56	2. crypto_decode_761x1531	32.4	32.93
Total	1,704.4	100.00	Total	98.4	100.00
NTRULPrime761 - Decapsulation					
1. Rq_mult_small x3	1,753.6	76.60	**1.1 Short_fromlist**	71.4	54.20
2. Short_fromlist	225.9	9.87	**1.2 expand x2**		
3. Expand x2	214.9	9.39	**1.3 Rq_mult_small x3**		
3. crypto_decode_761x1531 x2	45.4	1.98	**1.4 Hash x3**		
4. Hash x3	31.0	1.35	**1.5 crypto_encode_761x1531round**		
5. crypto_encode_761x1531round	4.3	0.19	**1.6 Others**		
Others	14.2	0.62	2. crypto_decode_761x1531 x2	60.3	45.80
Total	2,289.4	100.00	Total	131.7	100.00
NTRULPrime653 - Encapsulation					
1. Rq_mult_small x2	934.0	67.19	**1.1 Short_fromlist**	58.6	67.90
2. Short_fromlist	190.0	13.67	**1.2 Expand x2**		
3. Expand x2	183.6	13.21	**1.3 Rq_mult_small x2**		
4. Hash x4	48.2	3.46	**1.4 Hash x4**		
5. crypto_decode_653x1541	22.7	1.64	**1.5 crypto_encode_653x1541round**		
6. crypto_encode_653x1541round	3.6	0.26	**1.6 Others**		
Others	8.1	0.58	2. crypto_decode_653x1541	27.7	32.10
Total	1,390.2	100.00	Total	86.3	100.00
NTRULPrime653 - Decapsulation					
1. Rq_mult_small x3	1,400.9	75.50	**1.1 Short_fromlist**	64.2	55.60
2. Short_fromlist	187.9	10.13	**1.2 Expand x2**		
3. Expand x2	183.7	9.90	**1.3 Rq_mult_small x3**		
4. crypto_decode_653x1541 x2	38.8	2.09	**1.4 Hash x3**		
5. Hash x3	27.6	1.49	**1.5 crypto_encode_653x1541round**		
6. crypto_encode_653x1541round	3.7	0.20	**1.6 Others**		
Others	12.9	0.70	2. crypto_decode_653x1541 x2	51.3	44.40
Total	1,855.5	100.00	Total	115.5	100.00

Table 36: Results of profiling Streamlined NTRU Prime

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
StreamlinedNTRUPrime857 - Encapsulation					
1. Rq_mult_small	724.5	63.36	**1.1 crypto_sort_uint32**	60.1	62.95
2. crypto_sort_uint32	259.7	22.71	**1.2 Hash x5**		
3. Hash x5	71.9	6.29	**1.3 Rq_mult_small**		
4. Rq_decode	30.4	2.66	**1.4 Round_and_encode**		
5. Round_and_encode	8.2	0.72	**1.5 Others**		
Others	48.8	4.27	2. Rq_decode	35.4	37.05
Total	1,143.5	100.00	Total	95.5	100.00
StreamlineNTRUPrime857 - Decapsulation					
1. R3_mult	1,019.4	39.43	**1.1 Hash x4**	57.3	48.39
2. Rq_mult_small x2	1,448.9	56.05	**1.2 Rq_mult_small x2**		
3. Hash x4	42.1	1.63	**1.3 R3_mult**		
4. Rq_decode	27.9	1.08	**1.4 Others**		
5. Rounded_decode	28.1	1.09	2. Rq_decode	33.0	27.90
Others	18.9	0.73	3. Rounded_decode	28.1	23.71
Total	2,585.2	100.00	Total	118.4	100.00
StreamlineNTRUPrime761 - Encapsulation					
1. Rq_mult_small	584.6	61.65	**1.1 crypto_sort_uint32**	56.3	63.97
2. crypto_sort_uint32	223.8	23.61	**1.2 Hash x5**		
3. Hash x5	62.5	6.59	**1.3 Rq_mult_small**		
4. Rq_decode	27.0	2.85	**1.4. Round_and_encode**		
5. Round_and_encode	7.7	0.81	**1.5 Others**		
Others	42.7	4.50	2. Rq_decode	31.7	36.03
Total	948.2	100.00	Total	88.0	100.00
StreamlineNTRUPrime761- Decapsulation					
1. R3_mult	816.2	39.06	**1.1 Hash x4**	53.3	59.35
2. Rq_mult_small x2	1,169.4	55.96	**1.2 Rq_mult_small x2**		
3. Hash x4	35.8	1.72	**1.3 R3_mult**		
4. Rq_decode	24.5	1.17	**1.4 Others**		
5. Rounded_decode	25.8	1.24	2. Rq_decode	32.3	35.97
Others	17.8	0.85	3. Rounded_decode	4.2	4.68
Total	2,089.6	100.00	Total	89.8	100.00
StreamlineNTRUPrime653 - Encapsulation					
1. Rq_mult_small	467.0	60.19	**1.1 crypto_sort_uint32**	52.3	65.60
2. crypto_sort_uint32	185.5	23.90	**1.2 Hash x5**		
3. Hash x5	54.8	7.06	**1.3 Rq_mult_small**		
4. Rq_decode	24.2	3.11	**1.4 Round_and_encode**		
5. Round_and_encode	6.4	0.82	**1.5 Others**		
Others	38.2	4.92	2. Rq_decode	27.4	34.40
Total	775.9	100.00	Total	79.7	100.00
StreamlineNTRUPrime653- Decapsulation					
1. R3_mult	617.3	37.58	**1.1 Hash x4**	51.0	63.88
2. Rq_mult_small x2	933.7	56.85	**1.2 Rq_mult_small x2**		
3. Hash x4	35.7	2.17	**1.3 R3_mult**		
4. Rq_decode	21.0	1.28	**1.4 Others**		
5. Rounded_decode	22.4	1.37	2. Rq_decode	25.2	31.61
Others	12.4	0.76	3. Rounded_decode	3.6	4.51
Total	1,642.5	100.00	Total	79.8	100.00

Table 37: Results of profiling Round5

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
R5ND_CCA_5KEM_0d - Encapsulation					
1. r5_cpa_pke_encrypt	290.8	86.42	**1.1. r5_cpa_pke_encrypt**	30.1	95.20
2. hash	44.2	13.13	**1.2. hash**		
3. randombytes	1.5	0.45	2. randombytes	1.5	4.80
Total	336.5	100.00	Total	31.6	100.00
R5ND_CCA_5KEM_0d - Decapsulation					
1. r5_cpa_pke_encrypt	287.1	69.05	**1.1 r5_cpa_pke_encrypt**	36.8	100.00
2. r5_cpa_pke_decrypt	83.6	20.11	**1.2 r5_cpa_pke_decrypt**		
3. hash & verify	45.1	10.84	**1.3 hash & verify**		
Total	415.8	100.00	Total	36.8	100.00
R5ND_CCA_3KEM_0d - Encapsulation					
1. r5_cpa_pke_encrypt	211.4	86.27	**1.1. r5_cpa_pke_encrypt**	22.6	95.72
2. hash	32.6	13.32	**1.2. hash**		
3. randombytes	1.0	0.41	2. randombytes	1.0	4.28
Total	245.0	100.00	Total	23.6	100.00
R5ND_CCA_3KEM_0d - Decapsulation					
1. r5_cpa_pke_encrypt	208.2	67.27	**1.1 r5_cpa_pke_encrypt**	27.6	100.00
2. r5_cpa_pke_decrypt	67.5	21.80	**1.2 r5_cpa_pke_decrypt**		
3. hash & verify	33.9	10.94	**1.3 hash & verify**		
Total	309.5	100.00	Total	27.6	100.00
R5ND_CCA_1KEM_0d - Encapsulation					
1. r5_cpa_pke_encrypt	133.9	86.69	**1.1. r5_cpa_pke_encrypt**	16.9	94.46
2. hash	19.6	12.67	**1.2. hash**		
3. randombytes	1.0	0.64	2. randombytes	1.0	5.54
Total	154.5	100.00	Total	17.9	100.00
R5ND_CCA_1KEM_0d - Decapsulation					
1. r5_cpa_pke_encrypt	130.3	67.61	**1.1 r5_cpa_pke_encrypt**	19.7	100.00
2. r5_cpa_pke_decrypt	41.7	21.62	**1.2 r5_cpa_pke_decrypt**		
3. hash & verify	20.8	10.77	**1.3 hash & verify**		
Total	192.7	100.00	Total	19.7	100.00
R5ND_CCA_5KEM_5d - Encapsulation					
1. r5_cpa_pke_encrypt	372.0	91.87	**1.1. r5_cpa_pke_encrypt**	26.4	94.55
2. hash	31.4	7.76	**1.2. hash**		
3. randombytes	1.5	0.38	2. randombytes	1.5	5.45
Total	404.9	100.00	Total	27.9	100.00
R5ND_CCA_5KEM_5d - Decapsulation					
1. r5_cpa_pke_encrypt	372.0	69.24	**1.1 r5_cpa_pke_encrypt**	32.5	100.00
2. r5_cpa_pke_decrypt	132.3	24.63	**1.2 r5_cpa_pke_decrypt**		
3. hash & verify	32.9	6.12	**1.3 hash & verify**		
Total	537.2	100.00	Total	32.5	100.00
R5ND_CCA_3KEM_5d - Encapsulation					
1. r5_cpa_pke_encrypt	214.3	88.83	**1.1. r5_cpa_pke_encrypt**	19.3	95.04
2. hash	25.9	10.75	**1.2. hash**		
3. randombytes	1.0	0.42	2. randombytes	1.0	4.96
Total	241.3	100.00	Total	20.4	100.00
R5ND_CCA_3KEM_5d - Decapsulation					
1. r5_cpa_pke_encrypt	214.2	67.19	**1.1 r5_cpa_pke_encrypt**	23.3	100.00
2. r5_cpa_pke_decrypt	78.5	24.63	**1.2 r5_cpa_pke_decrypt**		
3. hash & verify	26.1	8.18	**1.3 hash & verify**		
Total	318.8	100.00	Total	23.3	100.00
R5ND_CCA_1KEM_5d - Encapsulation					
1. r5_cpa_pke_encrypt	111.7	88.59	**1.1. r5_cpa_pke_encrypt**	13.5	93.15
2. hash	13.4	10.62	**1.2. hash**		
3. randombytes	1.0	0.79	2. randombytes	1.0	6.85
Total	126.1	100.00	Total	14.4	100.00
R5ND_CCA_1KEM_5d - Decapsulation					
1. r5_cpa_pke_encrypt	111.8	64.72	**1.1 r5_cpa_pke_encrypt**	16.0	100.00
2. r5_cpa_pke_decrypt	46.7	27.06	**1.2 r5_cpa_pke_decrypt**		
3. hash & verify	14.2	8.22	**1.3 hash & verify**		
Total	172.7	100.00	Total	16.0	100.00

Table 38: Results of profiling for Saber

Function	**Time [us]**	**Time [%]**	**Function**	**Time [us]**	**Time [%]**
Software			**Software/Hardware**		
FireSaber - Encapsulation					
1. MatrixVectorMul	815.84	69.08%	**1.1 MatrixVectorMul**	55.09	84.43%
2. InnerProduct	204.44	17.31%	**1.2 InnerProduct**		
3. GenMatrix	92.93	7.87%	**1.3 GenMatrix**		
4. Hash	45.10	3.82%	**1.4 Hash**		
5. GenSecret	12.50	1.06%	**1.5 GenSecret**		
Others	10.16	0.86%	Others	10.16	15.57%
Total	1,180.97	100.00%	Total	65.25	100.00%
FireSaber - Decapsulation					
1. MatrixVectorMul	816.34	59.31%	**1.1 MatrixVectorMul**	55.14	71.55%
2. InnerProduct x2	408.14	29.65%	**1.2 InnerProduct x2**		
3. GenMatrix	92.99	6.76%	**1.3 GenMatrix**		
4. Hash	24.49	1.78%	**1.4 Hash**		
5. GenSecret	12.53	0.91%	**1.5 GenSecret**		
Others	21.92	1.59%	Others	21.92	28.45%
Total	1,376.41	100.00%	Total	77.06	100.00%
Saber - Encapsulation					
1. MatrixVectorMul	458.94	63.55%	**1.1 MatrixVectorMul**	49.15	86.36%
2. InnerProduct	153.19	21.21%	**1.2 InnerProduct**		
3. GenMatrix	53.29	7.38%	**1.3 GenMatrix**		
4. Hash	37.98	5.26%	**1.4 Hash**		
5. GenSecret	10.97	1.52%	**1.5 GenSecret**		
Others	7.76	1.07%	Others	7.76	13.64%
Total	722.13	100.00%	Total	56.91	100.00%
Saber - Decapsulation					
1. MatrixVectorMul	458.98	52.93%	**1.1 MatrixVectorMul**	48.15	74.47%
2. InnerProduct x2	306.52	35.35%	**1.2 InnerProduct x2**		
3. GenMatrix	53.29	6.15%	**1.3 GenMatrix**		
4. Hash	20.87	2.41%	**1.4 Hash**		
5. GenSecret	11.00	1.27%	**1.5 GenSecret**		
Others	16.51	1.90%	Others	16.51	25.53%
Total	867.17	100.00%	Total	64.66	100.00%
LightSaber - Encapsulation					
1. MatrixVectorMul	203.70	54.55%	**1.1 MatrixVectorMul**	43.36	88.49%
2. InnerProduct	102.26	27.38%	**1.2 InnerProduct**		
3. GenMatrix	23.67	6.34%	**1.3 GenMatrix**		
4. Hash	27.31	7.31%	**1.4 Hash**		
5. GenSecret	10.86	2.91%	**1.5 GenSecret**		
Others	5.64	1.51%	Others	5.64	11.51%
Total	373.44	100.00%	Total	49.00	100.00%
LightSaber - Decapsulation					
1. MatrixVectorMul	204.43	43.44%	**1.1 MatrixVectorMul**	41.27	78.55%
2. InnerProduct x2	204.80	43.52%	**1.2 InnerProduct x2**		
3. GenMatrix	23.67	5.03%	**1.3 GenMatrix**		
4. Hash	15.55	3.30%	**1.4 Hash**		
5. GenSecret	10.83	2.30%	**1.5 GenSecret**		
Others	11.27	2.40%	Others	11.27	21.45%
Total	470.55	100.00%	Total	52.54	100.00%

B Pseudocode of Implemented Algorithms

Below we show the pseudocode of all implemented KEMs, with parts offloaded to hardware marked with the gray background.

B.1 FrodoKEM

Algorithm 1 Pseudocode of FrodoKEM.Encaps [69]

Input: Public key $pk = seed_A\ ||b \in \{0,1\}^{len_{seed_A}+D.n.\bar{n}}$.
Output: Ciphertext $c_1||c_2 \in \{0,1\}^{(\bar{m}.n+\bar{m}.\bar{n})D}$ and shared secret **SS**$\in \{0,1\}^{len_{ss}}$.

```
Choose a uniformly random key $\mu \leftarrow s\ U(\{0,1\}^{len_\mu})$
Compute $\mathbf{pkh} \leftarrow$ SHAKE$(pk, len_{pkh})$
Generate pseudorandom values $seed_{SE}\ ||k \leftarrow$ SHAKE $(\mathbf{pkh}||\mu, len_{seed_{SE}} + len_k)$
Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, ..., r^{(2\bar{m}n+\overline{mn}-1)})) \leftarrow$
SHAKE$(0x96||seed_{SE}, 2\bar{m}n + \overline{mn}.len_x)$
Sample error matrix $\mathbf{S}$' $\leftarrow$ Frodo.SampleMatrix$((r^{(0)}, r^{(1)}, ..., r^{(\bar{m}n-1)})), \bar{m}, n, T_x)$
Sample error matrix $\mathbf{E}$' $\leftarrow$ Frodo.SampleMatrix$((r^{(\bar{m}n)}, r^{(\bar{m}n+1)}, ..., r^{(2\bar{m}n-1)})), \bar{m}, n, T_x)$
Generate $\mathbf{A} \leftarrow$ Frodo.Gen$(seed_A)$
Compute $\mathbf{B}$' $\leftarrow\mathbf{S}$'$\mathbf{A}+\ \mathbf{E}$'
Compute $\mathbf{c}_1 \leftarrow$ Frodo.Pack($\mathbf{B}$')
Sample error matrix $\mathbf{E}$" $\leftarrow$ Frodo.SampleMatrix$((r^{(2\bar{m}n)}, r^{(2\bar{m}n+1)}, ..., r^{(2\bar{m}n+\overline{mn}-1)})), \bar{m}, \bar{n}, T_x)$
Compute $\mathbf{B} \leftarrow$ Frodo.Unpack($\mathbf{b}$,n, $\bar{n}$)
Compute $\mathbf{V} \leftarrow\mathbf{S}$'$\mathbf{B}+\ \mathbf{E}$"
Compute $\mathbf{C} \leftarrow\mathbf{V}$ + Frodo.Encode$(\mu)$
Compute $\mathbf{c}_2 \leftarrow$ Frodo.Pack$(\mathbf{C})$
Compute $\mathbf{ss} \leftarrow$ SHAKE$(\mathbf{c}_1||\mathbf{c}_2||\mathbf{k}, ken_{ss})$
return ciphertext $(\mathbf{c}_1||\mathbf{c}_2$ and shared secret $\mathbf{ss}$
```

Algorithm 2 Pseudocode of FrodoKEM.Decaps [69]

Input: Ciphertext $\mathbf{c}_1||\mathbf{c}_2 \in \{0,1\}^{\bar{m}.n+\bar{m}.\bar{n})D}$, secret key $sk' = (s||seed_A||\mathbf{b}, \mathbf{S}, \mathbf{pkh}) \in \{0,1\}^{len_s+len_{seed_A}+D.n.\bar{n}} \times Z_q^{n\times\bar{n}} \times \{0,1\}^{len_{pkh}}$.
Output: Shared secret $\mathbf{ss} \in \{0,1\}^{len_{ss}}$.

```
$\mathbf{B}$' ← Frodo.Unpack($c_1$)
$\mathbf{C}$ ← Frodo.Unpack($c_2$)
Compute $\mathbf{M} \leftarrow \mathbf{C}$ - $\mathbf{B}$'$\mathbf{S}$
Compute $\mu$' ← Frodo.Decode($\mathbf{M}$)
Parse $pk \leftarrow seed_A$ || $\mathbf{b}$
Generate pseudorandom values $seed_{SE}{}' \; ||k' \leftarrow$ SHAKE $(\mathbf{pkh}||\mu', len_{seed_{SE}} + len_k)$
Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, ..., r^{(2\bar{m}n+\overline{mn}-1)})) \leftarrow$ SHAKE$(0x96||seed'_{se}, 2\bar{m}n + \overline{mn}.len_x)$
Sample error matrix $\mathbf{S}$' ← Frodo.SampleMatrix$((r^{(0)}, r^{(1)}, ..., r^{(\bar{m}n-1)})), \bar{m}, n, T_x)$
Sample error matrix $\mathbf{E}$' ← Frodo.SampleMatrix$((r^{(\bar{m}n)}, r^{(\bar{m}n+1)}, ..., r^{(2\bar{m}n-1)})), \bar{m}, n, T_x)$
Generate $\mathbf{A}$ ← Frodo.Gen($seed_A$)
Compute $\mathbf{B}$" ←$\mathbf{S}$'$\mathbf{A}$+ $\mathbf{E}$'
Sample error matrix $\mathbf{E}$" ← Frodo.SampleMatrix$((r^{(2\bar{m}n)}, r^{(2\bar{m}n+1)}, ..., r^{(2\bar{m}n+\overline{mn}-1)})), \bar{m}, \bar{n}, T_x)$
Compute $\mathbf{B}$ ← Frodo.Unpack($\mathbf{b}$,n, $\bar{n}$)
Compute $\mathbf{V}$ ←$\mathbf{S}$'$\mathbf{B}$+ $\mathbf{E}$"
Compute C' ←$\mathbf{V}$ + Frodo.Encode($\mu'$)
if B' || C = B" || C' then
    return shared secret ss ← SHAKE $(c_1||c_2||k', len_{ss})$
else
    return shared secret ss ← SHAKE $(c_1||c_2||s, len_{ss})$
end if
```

B.2 LAC

Algorithm 3 LAC-CCA Key Generation [52]

Input: Random *seed*
Output: $pk = (seed_{\vec{a}}, \vec{b}), sk = (\vec{s}, pk)$

```
$(seed_{\vec{a}}, seed_{\vec{s}}, seed_{\vec{e}}) \leftarrow \mathsf{GenSeed}(seed)$
$\vec{a} \leftarrow \mathsf{UniformSampl}(seed_{\vec{a}}) \in R_q$
$\vec{s} \leftarrow \mathsf{CBDSampl}(seed_{\vec{s}})$
$\vec{e} \leftarrow \mathsf{CBDSampl}(seed_{\vec{e}})$
$\vec{b} \leftarrow \vec{a}\vec{s} + \vec{e} \in R_q$
$pk = (seed_{\vec{a}}, \vec{b})$
$sk = (\vec{s}, pk = (seed_{\vec{a}}, \vec{b}))$
```

Algorithm 4 LAC-CCA Encapsulation [52]

Input: $pk = (seed_{\vec{a}}, \vec{b})$, message $\vec{m}$
Output: A ciphertext $\vec{c}$ and a session key $\vec{ss}$

```
$seed \leftarrow \mathsf{GenSeed}(\vec{m}, pk)$
$seeds \leftarrow \mathsf{GenSeed}(seed)$
$\vec{a} \leftarrow \mathsf{UniformSampl}(seed_{\vec{a}}) \in R_q$
$(\vec{r}, \vec{e}_1, \vec{e}_2) \leftarrow \mathsf{CBDSampl}(seeds)$
$\vec{c}_1 \leftarrow \vec{a}\vec{r} + \vec{e}_1 \in R_q$
$\vec{m}' \leftarrow \mathsf{BCH.Enc}(\vec{m}) \in \{0,1\}^{l_v}$
$\vec{m}'' \leftarrow \mathsf{D2.Enc}(\vec{m}') \in \mathbb{Z}_q^{l_v}$
$\vec{c}_2 \leftarrow \mathsf{Compress}((\vec{b}\vec{r})_{l_v} + \vec{e}_2 + \vec{m}'')$
$\vec{c} \leftarrow (\vec{c}_1, \vec{c}_2)$
$\vec{ss} \leftarrow \mathsf{Hash}(\vec{m}, \vec{c})$
```

Algorithm 5 LAC-CCA Decapsulation [52]

Input: $sk = (\vec{s}, pk = (seed_{\vec{a}}, \vec{b}))$, $\vec{c} = (\vec{c}_1, \vec{c}_2)$
Output: A session key $\vec{ss}$

```
$\vec{u} \leftarrow \vec{c}_1\vec{s} \in R_q$
$\vec{m}'' \leftarrow \mathsf{Decompress}(\vec{c}_2) - (\vec{u})_{l_v} \in \mathbb{Z}_q^{l_v}$
$\vec{m}' \leftarrow \mathsf{D2.Dec}(\vec{m}'') \in \{0,1\}^{l_v}$
$\vec{m} \leftarrow \mathsf{BCH.Dec}(\vec{m}')$
$seed \leftarrow \mathsf{GenSeed}(\vec{m}, pk)$
$seeds \leftarrow \mathsf{GenSeed}(seed)$
$\vec{a} \leftarrow \mathsf{UniformSampl}(seed_{\vec{a}}) \in R_q$
$(\vec{r}, \vec{e}_1, \vec{e}_2) \leftarrow \mathsf{CBDSampl}(seeds)$
$\underline{\vec{c}_1} \leftarrow \vec{a}\vec{r} + \vec{e}_1 \in R_q$
$\underline{\vec{m}'} \leftarrow \mathsf{BCH.Enc}(\vec{m}) \in \{0,1\}^{l_v}$
$\underline{\vec{m}''} \leftarrow \mathsf{D2.Enc}(\underline{\vec{m}'}) \in \mathbb{Z}_q^{l_v}$
$\underline{\vec{c}_2} \leftarrow \mathsf{Compress}((\vec{b}\vec{r})_{l_v} + \vec{e}_2 + \underline{\vec{m}''})$
$\underline{\vec{c}} \leftarrow (\underline{\vec{c}_1}, \underline{\vec{c}_2})$
if $\vec{c} = \underline{\vec{c}}$ then
    $\vec{ss} \leftarrow \mathsf{Hash}(\vec{m}, \vec{c})$
else
    $\vec{ss} \leftarrow \mathsf{Hash}(\mathsf{Hash}(sk), \vec{c})$
end if
```

B.3 Kyber

Algorithm 6 Pseudocode of KYBER.CCAKEM.Enc(pk) [8]

```
Input: Public key $pk \in \mathcal{B}^{12.k.n/8+32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u.k.n/8+d_v.n/8}$
Output: Shared key $K \in \mathcal{B}^*$
$m \leftarrow \mathcal{B}^{32}$
$m \leftarrow \mathrm{H}(m)$
$(\bar{K}, r) := \mathrm{G}(m||\mathrm{H}(pk))$
$c :=$ KYBER.CPAPKE.Enc$(pk, m, r)$
$K := \mathrm{KDF}(\bar{K}||\mathrm{H}(c))$
return $(c, K)$
```

Algorithm 7 Pseudocode of KYBER.CCAKEM.Dec(c, sk) [8]

```
Input: Ciphertext $c \in \mathcal{B}^{d_u.k.n/8+d_v.n/8}$
Input: Secret key $sk \in \mathcal{B}^{24.k.n/8+96}$
Output: Shared key $K \in \mathcal{B}^*$
$pk := sk + 12.k.n/8$
$h := sk + 24.k.n/8 + 32 \in \mathcal{B}^{32}$
$z := sk + 24.k.n/8 + 64$
$m' :=$ KYBER.CPAPKE.Dec(s, (u, $v$))
$(\bar{K}', r') := \mathrm{G}(m'||h)$
$c' :=$ KYBER.CPAPKE.Enc$(pk, m', r')$
if $c = c'$ then
    return $K := \mathrm{KDF}(\bar{K}'||\mathrm{H}(c))$
else
    return $K := \mathrm{KDF}(z||\mathrm{H}(c))$
end if
return $K$
```

Algorithm 8 Pseudocode of KYBER.CPAPKE.Enc(pk, m, r): encryption [8]

Input: Public key $pk \in \mathcal{B}^{12.k.n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u.k.n/8+d_v.n/8}$

```
$N := 0$
$\hat{t} := \text{Decode}_{12}(pk)$
$\rho := pk + 12.k.n/8$
for $i$ from 0 to $k-1$ do
    for $j$ from 0 to $k-1$ do
        $\hat{A}^T[i][j] := \text{Parse}(\text{XOF}(\rho, i, j))$
    end for
end for
for $i$ from 0 to $k-1$ do
    $\text{r}[i] := \text{CBD}_\eta(\text{PRF}(r, N))$
    $N := N + 1$
end for
for $i$ from 0 to $k-1$ do
    $\text{e}_1[i] := \text{CBD}_\eta(\text{PRF}(r, N))$
    $N := N + 1$
end for
$e_2 := \text{CBD}_\eta(\text{PRF}(r, N))$
$\hat{r} := \text{NTT}(\text{r})$
$\text{u} := \text{NTT}^{-1}(\hat{A}^T \circ \hat{r}) + \text{e}_1$
$v := \text{NTT}^{-1}(\hat{t}^T \circ \hat{r}) + \text{e}_2 + \text{Decompress}_q(\text{Decode}_1(m), 1)$
$c_1 := \text{Encode}_{d_u}(\text{Compress}_q(\text{u}, d_u))$
$c_2 := \text{Encode}_{d_v}(\text{Compress}_q(\text{v}, d_v))$
return $c = (c_1 || c_2)$
```

Algorithm 9 Pseudocode of KYBER.CPAPKE.Dec(sk, c): decryption [8]

Input: Secret key $sk \in \mathcal{B}^{12.k.n/8}$
Input: Ciphertext $c \in \mathcal{B}^{d_u.k.n/8+d_v.n/8}$
Output: Message $m \in \mathcal{B}^{32}$

```
$\text{u} := \text{Decompress}_q(\text{Decode}_{d_u}(c), d_u)$
$v := \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u.k.n/8), d_v)$
$\hat{s} := \text{Decode}_{12}(sk)$
$m := \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{s}^T \circ \text{NTT}(\text{u})), 1))$
return $m$
```

B.4 NewHope

Algorithm 10 Pseudocode of NewHope-CCA-KEM Encapsulation [2]

```
function NewHope-CCA-KEM.Encaps($pk$)
    $coin \xleftarrow{\$} \{0,\dots,255\}^{32}$
    $\mu \leftarrow \mathrm{SHAE256}(32, coin) \in \{0,\dots,255\}^{32}$
    $K||coin'||d \leftarrow \mathrm{SHAE256}(96, \mu||\mathrm{SHAE256}(32, pk)) \in \{0,\dots,255\}^{32+32+32}$
    $c \leftarrow \textsc{NewHope-CPA-PKE.Encrypt}(pk, \mu, coins')$
    $ss \leftarrow \mathrm{SHAKE256}(32, K||\mathrm{SHAKE256}(32, c||d))$
    return $(\bar{c} = c||d, ss)$
```

Algorithm 11 Pseudocode of NewHope-CCA-KEM Decapsulation [2]

```
function NewHope-CCA-KEM.Decaps($\bar{c}, \bar{sk}$)
    $c||d \leftarrow \bar{c} \in \{0,\dots,255\}^{3n/8+7n/4+32}$
    $sk||pk||h|s \leftarrow \bar{sk} \in \{0,\dots,255\}^{7n/4+7n/4+32+32+32}$
    $\mu' \leftarrow \mathrm{CPA.Decrypt}(c, sk)$
    $K'||coin''||d' \leftarrow \mathrm{SHAKE256}(96, \mu'||h) \in \{0,\dots,255\}^{32+32+32}$
    if $c = \mathrm{NewHope\text{-}CPA\text{-}PKE.Encrypt}(pk, \mu', coin'')$ and $d = d'$ then
        $fail \leftarrow 0$
    else
        $fail \leftarrow 1$
    $K \leftarrow K'$
    $K \leftarrow s$
    return $(ss = \mathrm{SHAKE256}(32, K_{fail}||\mathrm{SHAKE256}(32, c||d)))$
```

Algorithm 12 Pseudocode of NewHope-CPA-PKE Encryption [2]

```
function NewHope-CPA-PKE.Encrypt($pk \in \{0,\dots,255\}^{7n/4+32}, \mu \in \{0,\dots,255\}^{32}$,
$coin \in \{0,\dots,255\}^{32}$)
    $(\hat{b}, publicseed) \leftarrow \mathrm{DecodePk}(pk)$
    $\hat{a} \leftarrow \mathrm{GenA}(publicseed)$
    $s' \leftarrow \mathrm{BitRev}(\mathrm{Sample}(coin, 0))$
    $e' \leftarrow \mathrm{BitRev}(\mathrm{Sample}(coin, 1))$
    $e'' \leftarrow \mathrm{Sample}(coin, 2)$
    $\hat{t} \leftarrow \mathrm{NTT}(s')$
    $\hat{u} \leftarrow \hat{a} \circ \hat{t} + \mathrm{NTT}(e')$
    $v \leftarrow \mathrm{Encode}(\mu)$
    $v' \leftarrow \mathrm{NTT}^{-1}(\hat{b} \circ \hat{t}) + e'' + v$
    $h \leftarrow \mathrm{Compress}(v')$
    return $c = \mathrm{EncodeC}(\hat{u}, h)$
```

Algorithm 13 Pseudocode of NewHope-CPA-PKE Decryption [2]

```
function NewHope-CPA-PKE.Decrypt($c \in \{0, \ldots, 255\}^{7n/4+3n/8}, sk \in \{0, \ldots, 255\}^{7n/4}$)
    $(\hat{u}, h) \leftarrow \mathrm{DecodeC}(c)$
    $\hat{s} \leftarrow \mathrm{DecodePoly}(sk)$
    $v' \leftarrow \mathrm{Decompress}(h)$
    $\mu = \mathrm{Decode}(v' - \mathrm{NTT}^{-1}(\hat{u} \circ \hat{s}))$
    return $\mu$
```

B.5 NTRU-HPS and NTRU-HRSS

Algorithm 14 Pseudocode of NTRU KEM Encapsulate(h) [71]

```
$coins \leftarrow_{\$} \{0, 1\}^{256}$
$(r, m) \leftarrow$ Sample_rm $(coins)$
$c \leftarrow$ Encrypt $(h, (r, m))$
$k \leftarrow H_1(r, m)$
return $(c, k)$
```

Algorithm 15 Pseudocode of NTRU KEM Decapsulate($(f, f_p, h_q, s), c$) [71]

```
$(r, m, fail) \leftarrow \mathrm{Decrypt}((f, f_p, h_q), c)$
$k_1 \leftarrow H_1(r, M)$
$k_2 \leftarrow H_2(s, c)$
if $fail \ = \ 0$ then
    return $k_1$
else
    return $K_2$
end if
```

Algorithm 16 Pseudocode of NTRU DPKE Encrypt$(h,(r,m)))$ [71]

```
$m' \leftarrow \text{Lift}(m)$
$c \leftarrow (r \cdot h + m')\ mod\ (q, \Phi_1\Phi_n)$
return $c$
```

Algorithm 17 Pseudocode of NTRU DPKE Decrypt$((f, f_p, h_p), c)$ [71]

```
if $c \neq 0\ (mod\ (q, \Phi_1))$ then
    return $(0, 0, 1)$
end if
$a \leftarrow (c \cdot f)\ mod\ (q, \Phi_1\Phi_n)$
$m \leftarrow (a \cdot f_p)\ mod\ (3, \Phi_n)$
$m' \leftarrow \text{Lift}\ (m)$
$r \leftarrow ((c - m') \cdot h_q)\ mod\ (q, \Phi_n)$
if $(r, m) \in \mathcal{L}_r \times \mathcal{L}_m$ then
    return $(r, m, 0)$
else
    return $(0, 0, 1)$
end if
```

B.6 Streamlined NTRU Prime and NTRULPRime

Algorithm 18 Pseudocode of Encapsulation in Streamlined NTRU Prime and NTRULPRime [70]

Input: $\underline{K} \in \underline{\mathsf{PublicKeys}}$.
Output: $\mathsf{Ciphertexts}' \times \mathsf{SessionKeys}' = \underline{\mathsf{Ciphertexts}} \times \mathsf{Confirm} \times \mathsf{SessionKeys}'$.

```
Decode $\underline{K}$, obtaining $K \in \mathsf{PublicKeys}$.
Generate a uniform random $r \in \mathsf{Inputs}$.
Encode $r$ as a string $\underline{r} \in \mathsf{Inputs}$.
Compute $c = \mathsf{Encrypt}(r, K) \in \mathsf{Ciphertexts}$.
Encode $c$ as a string $\underline{c} \in \mathsf{Ciphertexts}$.
Compute $C = (\underline{c}, \mathsf{HashConfirm}(\underline{r}, \underline{K})) \in \mathsf{Ciphertexts} \times \mathsf{Confirm}$
Return $(C, \mathsf{HashSession}(1, \underline{r}, C))$.
```

Algorithm 19 Pseudocode of Decapsulation in Streamlined NTRU Prime and NTRULPRime [70]

Input: $C = (\underline{c}, \gamma) \in \underline{\mathsf{Ciphertexts}} \times \mathsf{Confirm}$ and $(\underline{k}, \underline{K}, \rho) \in \underline{\mathsf{SecretKeys}} \times \underline{\mathsf{PublicKeys}} \times \underline{\mathsf{Inputs}}$
Output: $\mathsf{SessionKeys}$

```
Decode $\underline{c}$, obtaining $c \in \mathsf{Ciphertexts}$.
Decode $\underline{k}$, obtaining $k \in \mathsf{SecretKeys}$.
Compute $r' = \mathsf{Decrypt}(c, k) \in \mathsf{Inputs}$.
Encode $r'$ as a string $\underline{r'} \in \mathsf{Inputs}$.
Compute $c' = \mathsf{Encrypt}(r', K) \in \mathsf{Ciphertexts}$.
Encode $c'$ as a string $\underline{c'} \in \mathsf{Ciphertexts}$.
Compute $C' = (\underline{c'}, \mathsf{HashConfirm}(\underline{r'}, \underline{K})) \in \mathsf{Ciphertexts} \times \mathsf{Confirm}$
If $C' = C$ then return $\mathsf{HashSession}(1, \underline{r}, C)$. Otherwise return $\mathsf{HashSession}(0, \rho, C)$.
(The choice between these two outputs is secret information.)
```

Algorithm 20 Pseudocode of Encryption in Streamlined NTRU Prime [70]

Input: $r \in \mathsf{Inputs}$ and $K = h \in \mathsf{PublicKeys}$.
Output: $c \in \mathsf{Ciphertexts}$.

```
Compute $hr \in \mathcal{R}/q$.
Return $c = \mathsf{Round}(hr)$.
```

Algorithm 21 Pseudocode of Decryption in Streamlined NTRU Prime [70]

Input: $c \in \mathsf{Ciphertexts}$ and $k = (f, v) \in \mathsf{Short} \times \mathcal{R}/3$.
Output: $r' \in \mathsf{Inputs}$.

```
Compute $3fc \in \mathcal{L}/q$.
View each coefficient of $3fc \in \mathcal{R}/q$ as an integer between $-(q-1)/2$ and $(q-1)/2$, and
then reduce modulo 3, obtaining a polynomial $e \in \mathcal{R}/3$.
Multiply by $v \in \mathcal{R}/3$.
Lift $ev \in \mathcal{R}/3$ to a small polynomial $r' \in \mathcal{R}$.
Return $r'$ if $r'$ has weight $w$. Otherwise output (1, 1, ..., 1, 0, 0,..., 0).
```

Algorithm 22 Pseudocode of Encryption in NTRU LPRime Expand [70]

Input: $r \in \mathsf{Inputs}$ and $K = (S, A) \in \mathsf{Seeds} \times \mathsf{Rounded}$.
Output: $c \in \mathsf{Ciphertexts}$.

```
Compute $G = \mathsf{Generator}(S)$.
Generate a uniform random $b \in \mathsf{HashShort}(r)$.
Compute $bG$ in $\mathcal{R}/q$.
Compute $bA$ in $\mathcal{R}/q$.
Compute $T = (T_0, T_1, ..., T_{I-1}) \in (\mathbb{Z}/\tau)^I$ as follows: $T_j = \mathsf{Top}((bA)_j + r_j(q-1)/2)$.
Return $c = (\mathsf{Round}(bG), T) \in \mathsf{Ciphertexts}$.
```

Algorithm 23 Pseudocode of Decryption in NTRU LPRime Expand [70]

Input: $c = (B, T) \in \mathsf{Rounded} \times (\mathbb{Z}/\tau)^I$ and $k = a \in \mathsf{SecretKeys}$.
Output: $r' \in \mathsf{Inputs}$.

```
Compute $aB$ in $\mathcal{R}/q$.
Compute $(r'_0, r'_1, ..., r'_{I-1}) \in \{0, 1\}$ as follows. View $\mathsf{Right}(T_j) - (aB)_j + 4w + 1 \in (\mathbb{Z}/q)$
as an integer between $-(q-1)/2$ and $(q-1)/2$. Then $r'_j$ is the sign bit of this integer:
1 if the integer is negative, otherwise 0.
Return $r' = (r'_0, r'_1, ..., r'_{I-1}) \in \mathsf{Inputs}$
```

B.7 Round5

Algorithm 24 Pseudocode of r5_cca_kem_encapsulate(pk) [72]

Parameters: Integers $p, t, q, n, d, \bar{m}, \bar{n}, \mu, b, k, f, \tau; \xi \in \{\Phi_{n+1}(x), x^{n+1} - 1\}$
Input: $pk \in \{0,1\}^k \times R_{n,p}^{d/n \times \bar{n}}$
Output: $ct = (\tilde{U}, v, g) \in R_{n,p}^{\bar{m} \times d/n} \times Z_t^{\mu} \times \{0,1\}^k, k \in \{0,1\}^k$

```
$m \xleftarrow{\$} \{0,1\}^k$
$(L, g, \rho) = G(m||pk)$
$(\tilde{U}, v) = r5\_cpa\_pke\_encrypt(pk, m, \rho)$
$ct = (\tilde{U}, v, g)$
$k = H(L||ct)$
return $(ct, k)$
```

Algorithm 25 Pseudocode of r5_cca_kem_decapsulate(ct,sk) [72]

Parameters: Integers $p, t, q, n, d, \bar{m}, \bar{n}, \mu, b, k, f, \tau; \xi \in \{\Phi_{n+1}(x), x^{n+1} - 1\}$
Input: $ct = (\tilde{U}, v, g) \in R_{n,p}^{\bar{m} \times d/n} \times Z_t^{\mu} \times \{0,1\}^k, sk = (sk_{CPA-PKE,y,pk}) \in \{0,1\}^k \times \{0,1\}^k \times (\{0,1\}^k \times R_{n,p}^{d/n \times \bar{n}})$
Output: $k \in \{0,1\}^k$

```
$m' = r5\_cpa\_pke\_decrypt(sk_{CPA-PKE,(\tilde{U},v))}$
$(L', g', \rho') = G(m'||pk)$
$(\tilde{U}', v') = r5\_cpa\_pke\_encrypt(pk, m', \rho')$
$ct' = (\tilde{U}', v', g')$
if $(ct = ct')$ then
    return $k = H(L'||ct))$
else
    return $k = H(y||ct))$
end if
```

Algorithm 26 Pseudocode of r5_cpa_pke_encrypt(pk) [72]

Parameters: Integers $p, t, q, n, d, \bar{m}, \bar{n}, \mu, b, k, f, \tau; \xi \in \{\Phi_{n+1}(x), x^{n+1} - 1\}$
Input: $pk = (\sigma, B) \in \{0,1\}^k \times R_{n,p}^{d/n \times \bar{n}}, m, \rho \in \{0,1\}^k$
Output: $ct = (\tilde{U}, v) \in R_{n,p}^{\bar{m} \times d/n} \times Z_t^{\mu}$

```
$A = f^{(\tau)}_{d.n}(c)$
$R = f_R(\rho)$
$U = R_{q \to p, h_2}(\langle A^T R \rangle_{\Phi_{n+1}})$
$\tilde{U} = U^T$
$v = \langle R_{p \to t, h_2}(Sample_{\mu}(\langle B^T R \rangle_{\xi})) \rangle_{+}$
$ct = (\tilde{U}, v)$
return $ct$
```

Algorithm 27 Pseudocode of r5_cpa_pke_decrypt(sk,ct) [72]

Parameters: Integers $p, t, q, n, d, \bar{m}, \bar{n}, \mu, b, k, f, \tau; \xi \in \{\Phi_{n+1}(x), x^{n+1} - 1\}$
Input: $sk \in \{0,1\}^k, ct = (\tilde{U}, v) \in R_{n,p}^{\bar{m} \times d/n} \times Z_t^{\mu}$
Output: $\hat{m} \in \{0,1\}^k$

```
$v_p = \frac{p}{t} b$
$S = f_s(sk)$
$U = \tilde{U}^T$
$y = R_{p \to b, h_3}(v_p - Sample_{\mu}((S^T(U + h_4 J))_{\xi}))$
return $\hat{m}$
```

B.8 Saber

Algorithm 28 Pseudocode of Saber.KEM.Encaps $(pk = (seed_A, b))$ [73]

```
$m \leftarrow u(\{0,1\}^{256})$
$(\hat{K}, r) = g(F(pk), m)$
$c = $ Saber.PKE.Enc$(pk, m; r)$
$K = H(\hat{K}, c)$
return $(c, K)$
```

Algorithm 29 Pseudocode of Saber.KEM.Decaps ($sk = (s, z, pkh), pk = (seed_A, b), c$) [73]

```
$m' = \text{Saber.PKE.Dec}(s, c)$
$(\hat{K}, r') = g(pkh, m')$
$c' = \text{Saber.PKE.Enc}(pk, m'; r')$
if $c = ct'$ then
    return $K = H(\hat{K}', c)$
else
    return $K = H(z, c)$
end if
```

Algorithm 30 Pseudocode of Saber.PKE.Enc ($pk = (seed_A, b), m \in R_2; r$) [73]

```
$A = gen(seed_A) \in R_q^{l \times l}$
$(\hat{K}, r') = g(pkh, m')$
if $r$ is not specified then
    $r = u(\{0, 1\}^{256})$
end if
$s' = \beta_\mu(R_q^{l \times l}; r)$
$b' = ((As' + h) \ mod \ q) >> (\epsilon_q - \epsilon_p) \in R_p^{l \times l}$
$v' = b^T(s' \ mod \ p) \in R_p$
$c_m = (v' + h_1 - 2^{\epsilon_p - 1} mmodp) >> (\epsilon_q - \epsilon_T) \in R_T$
return $c := (c_m, b')$
```

Algorithm 31 Pseudocode of Saber.PKE.Dec ($sk = s, c = (c_m, b')$) [73]

```
$v = b'^T(s \ mod \ p) \in R_p$
$m' = ((v - 2^{\epsilon_p} - \epsilon_T c_m + h_2) \ mod \ p) >> (\epsilon_p - 1) \in R_2$
return $m'$
```

C Speed-ups Compared to the Best Portable Implementations in C

In Table 39, for each investigated KEM and each major operation (Encapsulation and Decapsulation), we list the total execution time in software (for the optimized portable software implementations in C running on ARM Cortex-A53 of Zynq UltraScale+ MPSoC), the total execution time in software and hardware (after offloading the most time-consuming operations to hardware), and the obtained speed-up. The ARM processor runs at 1.2 GHz, DMA for the communication between the processor and the hardware accelerator at 200 MHz, and the hardware accelerators at the maximum frequencies, specific for the RTL implementations of each algorithm, listed in Table 23. All execution times were obtained through experimental measurements using the setup shown in Fig. 1. The speed-up for the software part offloaded to hardware itself is given in the column Accel. Speed-up. This speed-up is a ratio of the execution time of the accelerated portion in software (column Accel. SW [ms]) and the execution time of the accelerated portion in hardware, including all overheads (column Accel. HW [ms]). The last column indicates how big percentage of the software-only execution time was taken by an accelerated portion of the program. Links to the underlying software implementations are summarized in Table 22.

Table 39: Speed-ups Compared to the Best Portable Implementations in C

Algorithm	Parameter Set	Total SW [ms]	Total SW/HW [ms]	Total Speed-up	Accel. SW [ms]	Accel HW [ms]	Accel. Speed-up	SW part Sped up by HW [%]
				Encapsulation				
FrodoKEM	Frodo-640	16.192	1.223	13.2	15.322	0.353	43.5	94.62
FrodoKEM	Frodo-976	34.609	1.642	21.1	33.727	0.761	44.3	97.45
FrodoKEM	Frodo-1344	62.076	2.186	28.4	61.218	1.328	46.1	98.62
Kyber	Kyber_512	0.327	0.015	21.5	0.326	0.014	23.9	99.52
Kyber	Kyber_768	0.533	0.018	29.8	0.531	0.016	32.5	99.71
Kyber	Kyber_1024	0.784	0.022	35.5	0.783	0.021	38.2	99.80
LAC-v3a	LAC-128-v3a	0.333	0.016	22.0	0.332	0.015	22.2	99.70
LAC-v3a	LAC-192-v3a	0.564	0.021	28.0	0.562	0.020	28.3	99.73
LAC-v3a	LAC-256-v3a	0.906	0.024	41.0	0.905	0.022	40.6	99.83
LAC-v3b	LAC-128-v3b	0.315	0.014	23.0	0.314	0.013	23.4	99.68
LAC-v3b	LAC-192-v3b	0.530	0.018	31.0	0.529	0.017	31.4	99.72
LAC-v3b	LAC-256-v3b	0.870	0.021	44.0	0.868	0.020	44.4	99.83
NewHope	NewHope_512	0.348	0.015	23.1	0.346	0.013	25.7	99.55
NewHope	NewHope_1024	0.723	0.021	34.0	0.722	0.020	36.7	99.78
Round5	R5ND-CCA1KEM0d	0.155	0.018	8.6	0.154	0.017	9.1	99.36
Round5	R5ND-CCA3KEM0d	0.245	0.024	10.4	0.244	0.023	10.8	99.59
Round5	R5ND-CCA5KEM0d	0.337	0.032	10.6	0.335	0.030	11.1	99.55
Round5	R5ND_CCA1KEM5d	0.126	0.014	8.7	0.125	0.013	9.3	99.21
Round5	R5ND_CCA3KEM5d	0.241	0.020	11.9	0.240	0.019	12.4	99.58
Round5	R5ND_CCA5KEM5d	0.405	0.028	14.5	0.403	0.026	15.3	99.62
Saber	LightSaber-KEM	0.373	0.049	7.6	0.368	0.043	8.5	98.49
Saber	Saber-KEM	0.722	0.057	12.7	0.714	0.049	14.5	98.93
Saber	FireSaber-KEM	1.181	0.065	18.1	1.171	0.055	21.3	99.14
NTRU LPRime	kem/ntrulpr653	1.390	0.052	26.9	1.367	0.029	47.4	98.36
NTRU LPRime	kem/ntrulpr761	1.704	0.060	28.6	1.679	0.034	49.6	98.49
NTRU LPRime	kem/ntrulpr857	2.061	0.067	30.6	2.032	0.039	52.3	98.62
NTRU-HPS	ntruhps2048677	2.961	0.041	71.9	2.946	0.026	115.2	99.47
NTRU-HPS	ntruhps4096821	4.285	0.048	88.5	4.268	0.032	135.4	99.61
NTRU-HRSS	ntruhrss701	2.964	0.068	43.4	2.921	0.025	115.4	98.55
Str NTRU Prime	kem/sntrup653	0.776	0.049	16.0	0.752	0.024	30.9	96.89
Str NTRU Prime	kem/sntrup761	0.948	0.056	17.1	0.921	0.029	32.3	97.15
Str NTRU Prime	kem/sntrup857	1.143	0.063	18.0	1.113	0.033	33.7	97.34
				Decapsulation				
FrodoKEM	Frodo-640	16.192	1.321	12.3	15.214	0.343	44.4	93.96
FrodoKEM	Frodo-976	34.649	1.866	18.6	33.532	0.750	44.7	96.78
FrodoKEM	Frodo-1344	62.377	3.120	20.0	60.573	1.317	46.0	97.11
Kyber	Kyber_512	0.428	0.017	25.1	0.428	0.017	25.1	100.00
Kyber	Kyber_768	0.666	0.020	33.2	0.666	0.020	33.2	100.00
Kyber	Kyber_1024	0.950	0.025	38.5	0.950	0.025	38.5	100.00
LAC-v3a	LAC-128-v3a	0.463	0.017	27.1	0.463	0.017	27.1	100.00
LAC-v3a	LAC-192-v3a	0.782	0.024	32.9	0.782	0.024	32.9	100.00
LAC-v3a	LAC-256-v3a	1.378	0.027	51.3	1.378	0.027	51.3	100.00
LAC-v3b	LAC-128-v3b	0.440	0.015	28.4	0.440	0.015	28.4	100.00
LAC-v3b	LAC-192-v3b	0.741	0.021	36.0	0.741	0.021	36.0	100.00
LAC-v3b	LAC-256-v3b	1.332	0.024	55.2	1.332	0.024	55.2	100.00
NewHope	NewHope_512	0.426	0.016	26.4	0.426	0.016	26.4	100.00
NewHope	NewHope_1024	0.891	0.025	35.9	0.891	0.025	35.9	100.00
Round5	R5ND_CCA1KEM0d	0.193	0.020	9.8	0.19273	0.020	9.8	100.00
Round5	R5ND_CCA3KEM0d	0.309	0.028	11.2	0.30946	0.028	11.2	100.00
Round5	R5ND_CCA5KEM0d	0.416	0.037	11.3	0.4158	0.037	11.3	100.00
Round5	R5ND_CCA1KEM5d	0.173	0.016	10.8	0.17267	0.016	10.8	100.00
Round5	R5ND_CCA3KEM5d	0.319	0.023	13.7	0.31877	0.023	13.7	100.00
Round5	R5ND_CCA5KEM5d	0.537	0.033	16.5	0.53721	0.033	16.5	100.00
Saber	LightSaber-KEM	0.471	0.053	9.0	0.459	0.041	11.1	97.60
Saber	Saber-KEM	0.867	0.065	13.4	0.851	0.048	17.7	98.10
Saber	FireSaber-KEM	1.376	0.077	17.9	1.354	0.055	24.6	98.41
NTRU LPRime	kem/ntrulpr653	1.856	0.071	26.2	1.817	0.032	56.7	97.91
NTRU LPRime	kem/ntrulpr761	2.289	0.084	27.2	2.244	0.039	58.1	98.02
NTRU LPRime	kem/ntrulpr857	2.787	0.098	28.6	2.736	0.047	57.8	98.20
NTRU-HPS	ntruhps2048677	8.175	0.095	85.8	8.114	0.034	238.0	99.25
NTRU-HPS	ntruhps4096821	11.982	0.107	111.9	11.912	0.037	318.6	99.42
NTRU-HRSS	ntruhrss701	8.790	0.136	64.8	8.700	0.046	187.7	98.98
Str NTRU Prime	kem/sntrup653	1.643	0.067	24.6	1.599	0.023	68.2	97.36
Str NTRU Prime	kem/sntrup761	2.090	0.079	26.5	2.039	0.029	71.5	97.59
Str NTRU Prime	kem/sntrup857	2.585	0.087	29.8	2.529	0.031	82.0	97.84